Chapter 4. SOM IDL and the SOM Compiler

Contents

Chapter 4. SOM IDL and the SOM Compiler

This chapter first discusses how to define SOM classes and then describes the SOM Compiler. To allow a class of objects to be implemented in one programming language and used in another (that is, to allow a SOM class to be language neutral), the interface to objects of this class must be specified separately from the objects' implementation.

To summarize: As a first step, a file known as the .idl file is used to declare classes and their methods, using SOM's language-neutral Interface Definition Language (IDL). Next, the SOM Compiler is run on the .idl file to produce a template implementation file that contains stub method procedures for the new and overridden methods; this preliminary code corresponds to the computer language that will implement the class. Then, the class implementor fills in the stub procedures with code that implements the methods (or redefines overridden methods) and sets instance data. (This implementation process is the subject of Chapter 5, "Implementing Classes in SOM.") At this point, the implementation file can be compiled and linked with a client program that uses it (as described in Chapter 3, "Using SOM Classes in Client Programs").

Syntax for SOM IDL and the SOM Compiler are presented in this chapter, along with helpful information for using them correctly.

4.1 Interface vs Implementation

The *interface* to a class of objects contains the information that a client must know to use an object — namely, the names of its attributes and the signatures of its methods. The interface is described in a formal language independent of the programming language used to implement the object's methods. In SOM, the formal language used to define object interfaces is the **Interface Definition Language** (IDL), standardized by CORBA.

The *implementation* of a class of objects (that is, the procedures that implement methods and the variables used to store an object's state) is written in the implementor's preferred programming language. This language can be object-oriented (for instance, C++) or procedural (for instance, C).

A completely implemented class definition, then, consists of two main files:

- An IDL specification of the interface to instances of the class — the *interface definition file* (or .idl file) and
- Method procedures written in the implementor's language of choice — the *implementation file*.

The interface definition file has a .idl extension, as noted. The implementation file, however, has an extension specific to the language in which it is written. For example, implementations written in C have a .c extension, and implementations written in C++ have a .C (for AIX) or .cpp (for OS/2 or Windows) extension.

To assist users in implementing SOM classes, the SOMobjects Toolkit provides a SOM Compiler. The SOM Compiler takes as input an object interface definition file (the .idl file) and produces a set of *binding files* that make it convenient to implement and use a SOM class whose instances are objects that support the defined interface. The binding files and their purposes are as follows:

- An *implementation template* that serves as a guide for how the implementation file for the class should look. The class implementor fills in this template file with language-specific code to implement the methods that are available on the class' instances.
- *Header files* to be included (a) in the class's implementation file and (b) in client programs that use the class.

These binding files produced by the SOM Compiler bridge the gap between SOM and the object model used in object-oriented languages (such as C++), and they allow SOM to be used with non-object-oriented languages (such as C). The SOM Compiler currently produces binding files for the C and C++ programming languages. SOM can also be used with other programming languages; the bindings simply offer a more convenient programmer's interface to SOM. Vendors of other languages may also offer SOM bindings; check with your language vendor for possible SOM support.

The subsequent sections of this chapter provide full syntax for SOM IDL and the SOM Compiler.

4.2 SOM Interface Definition Language

This section describes the syntax of SOM's **Interface Definition Language (SOM IDL)**. SOM IDL complies with CORBA's standard for IDL; it also adds constructs specific to SOM. (For more information on the CORBA standard for IDL, see *The Common Object Request Broker: Architecture and Specification*, published by Object Management Group and x/Open.) The full grammar for SOM IDL is given in Appendix C. Instructions for converting existing OIDL-syntax files to IDL are given in Appendix B. The current section describes the syntax and semantics of SOM IDL using the following conventions:

Constants (words to be used literally, such as keywords) appear in **bold**.
User-supplied elements appear in *italics*.
{ } Groups related items together as a single item.
[ ] Encloses an optional item.
* Indicates zero or more repetitions of the preceding item.
+ Indicates one or more repetitions of the preceding item.
| Separates alternatives.
_ Within a set of alternatives, an underscore indicates the default, if defined.

IDL is a formal language used to describe object interfaces. Because, in SOM, objects are implemented as instances of classes, an IDL object interface definition specifies for a class of objects what methods (operations) are available, their return types, and their parameter types. For this reason, we often speak of an IDL specification for a class (as opposed to simply an object interface). Constructs specific to SOM discussed below further strengthen this connection between SOM classes, and the IDL language.

IDL generally follows the same lexical rules as C and C++, with some exceptions. In particular:

- IDL uses the ISO Latin-1 (8859.1) character set (as per the CORBA standard).
- White space is ignored except as token delimiters.
- C and C++ comment styles are supported.
- IDL supports standard C/C++ preprocessing, including macro substitution, conditional compilation, and source file inclusion.
- Identifiers (user-defined names for methods, attributes, instance variables, and so on) are composed of alphanumeric and underscore characters (with the first character alphabetic) and can be of arbitrary length, up to an operating-system limit of about 250 characters.
- Identifiers must be spelled consistently with respect to case throughout a specification.
- Identifiers that differ only in case yield a compilation error.
- There is a single name space for identifiers (thus, using the same identifier for a constant and a class name within the same naming scope, for example, yields a compilation error).
- Integer, floating point, character, and string literals are defined just as in C and C++.

The terms listed in Table 1 on the following page are reserved keywords and may not be used otherwise. Keywords must be spelled using upper- and lower-case characters exactly as shown in the table. For example, "void" is correct, but "Void" yields a compilation error.

A typical IDL specification for a single class, residing in a single .idl file, has the following form. (Also see the later section, "Defining multiple interfaces in a .idl file.") The order is unimportant, except that names must be declared (or forward referenced) before they are referenced. The subsequent topics of this section describe the requirements for these specifications:

Include directives	(optional)
Type declarations	(optional)
Constant declarations	(optional)
Exception declarations	(optional)
Interface declaration	(optional)
Module declaration	(optional)

Table 1. KEYWORDS FOR SOM IDL

any	**FALSE**	**readonly**
attribute	**float**	**sequence**
boolean	**implementation**	**short**
case	**in**	**string**
char	**inout**	**struct**
class	**interface**	**switch**
const	**long**	**TRUE**
context	**module**	**TypeCode**
default	**octet**	**typedef**
double	**oneway**	**unsigned**
enum	**out**	**union**
exception	**raises**	**void**

Include directives

The IDL specification for a class normally contains **#include** statements that tell the SOM Compiler where to find the interface definitions (the .idl files) for:

- Each of the class's parent (direct base) classes, and
- The class's metaclass (if specified).

The **#include** statements must appear in the above order. For example,
if class "C" has parents "foo" and "bar" and metaclass "meta",
then file "C.idl" must begin with the following **#include** statements:

```
#include <foo.idl>
#include <bar.idl>
#include <meta.idl>
```

As in C and C++, if a filename is enclosed in angle brackets (< >), the search for the file will begin in system-specific locations. If the filename appears in double quotation marks (" "), the search for the file will begin in the current working directory, then move to the system-specific locations.

Type and constant declarations

IDL specifications may include type declarations and constant declarations as in C and C++, with the restrictions/extensions described below. [Note: Readers unfamiliar with C might wish to consult *The C Programming Language* (2nd edition, 1988, Prentice Hall) by Brian W. Kernighan and Dennis M. Ritchie. See pages 36–40 for a discussion of type and constant declarations.]

IDL supports the following basic types (these basic types are also defined for C and C++ client and implementation programs, using the SOM bindings):

Integral types

IDL supports only the integral types **short**, **long**, **unsigned short**, and **unsigned long**, which represent the following value ranges:

short	$-2^{15} .. 2^{15}-1$
long	$-2^{31} .. 2^{31}-1$
unsigned short	$0 .. 2^{16}-1$
unsigned long	$0 .. 2^{32}-1$

Floating point types

IDL supports the **float** and **double** floating-point types. The **float** type represents the IEEE single-precision floating-point numbers; **double** represents the IEEE double-precision floating-point numbers. Note for Windows: Since returning floats and doubles by value may not be compatible across Windows compilers, client programs should return floats and doubles by reference.

Character type

IDL supports a **char** type, which represents an 8-bit quantity. The ISO Latin-1 (8859.1) character set defines the meaning and representation of graphic characters. The meaning and representation of null and formatting characters is the numerical value of the character as defined in the ASCII (ISO 646) standard. Unlike C/C++, type **char** cannot be qualified as signed or unsigned. (The **octet** type, below, can be used in place of unsigned char.)

Boolean type

IDL supports a **boolean** type for data items that can take only the values TRUE and FALSE.

Octet type

IDL supports an **octet** type, an 8-bit quantity guaranteed not to undergo conversion when transmitted by the communication system. The octet type can be used in place of the unsigned char type.

Any type

IDL supports an **any** type, which permits the specification of values of any IDL type. In the SOM C and C++ bindings, the **any** type is mapped onto the following **struct**:

```
typedef struct any {
     TypeCode _type;
     void *_value;
} any;
```

The "_value" member for an **any** type is a pointer to the actual value. The "_type" member is a pointer to an instance of a **TypeCode** that represents the type of the value. The **TypeCode** provides functions for obtaining information about an IDL type. Chapter 7, "The Interface Repository Framework," describes **TypeCode**s and their associated functions.

Constructed types

In addition to the above basic types, IDL also supports three **constructed** types: **struct, union**, and **enum.** The structure and enumeration types are specified in IDL the same as they are in C and C++ [Kernighan–Ritchie references: struct, p. 128; union, p. 147; enum, p. 39], with the following restrictions:

Unlike C/C++, recursive type specifications are allowed only through the use of the **sequence** template type (see below).

Unlike C/C++, structures, discriminated unions, and enumerations in IDL must be tagged. For example, "struct { int a; ... }" is an invalid type specification. The tag introduces a new type name.

In IDL, constructed type definitions need not be part of a **typedef** statement; furthermore, if they are part of a typedef statement, the tag of the struct must differ from the type name being defined by the typedef. For example, the following are valid IDL **struct** and **enum** definitions:

```
struct myStruct {
   long x;
   double y;
};                               /* defines type name myStruct */

enum colors { red, white, blue };  /* defines type name colors */
```

By contrast, the following IDL definitions are *not* valid:

```
typedef struct myStruct {        /*  NOT VALID  */
   long x;
   double y;
} myStruct;                      /* myStruct has been redefined */

typedef enum colors { red, white, blue } colors;  /* NOT VALID */
```

The valid IDL **struct** and **enum** definitions shown above are translated by the SOM Compiler into the following definitions in the C and C++ bindings, assuming they were declared within the scope of interface “Hello”:

```
typedef struct Hello_myStruct {  /* C/C++ bindings for IDL struct */
   long x;
   double y;
} Hello_myStruct;

typedef unsigned long Hello_colors; /* C/C++ bindings for IDL enum */
#define Hello_red 1UL
#define Hello_white 2UL
#define Hello_blue 3UL
```

When an enumeration is defined within an interface statement for a class, then within C/C++ programs, the enumeration names must be referenced by prefixing the class name. For example, if the *colors* enum, above, were defined within the interface statement for class *Hello*, then the enumeration names would be referenced as *Hello_red, Hello_white*, and *Hello_blue*. Notice the first identifier in an enumeration is assigned the value 1.

Note for Windows: Since returning structs may not be compatible across all compilers, client programs should, in general, return an object that contains a struct. For more information, see “General guidelines for class library designers” in Chapter 5, “Implementing Classes in SOM.”

All types and constants generated by the SOM Compiler are *fully qualified*. That is, prepended to them is the fully qualified name of the interface or module in which they appear. For example, consider the following fragment of IDL:

```
module M {
    typedef long long_t;
    module  N {
        typedef long long_t;
        interface I {
            typedef long long_t;
        };
    };
};
```

That specification would generate the following three types:

```
typedef long  M_long_t;
typedef long  M_N_long_t;
typedef long  M_N_I_long_t;
```

For programmer convenience, the SOM Compiler also generates shorter bindings, without the interface qualification. Consider the next IDL fragment:

```
module M {
    typedef long long_t;
    module  N {
        typedef short short_t;
        interface I {
            typedef char char_t;
        };
    };
};
```

In the C/C++ bindings of the preceding fragment, you can refer to “M_long_t” as “long_t”, to “M_N_short_t” as “short_t”, and to “M_N_I_char_t” as “char_t”. However, these shorter forms are available *only* when their interpretation is not ambiguous. Thus, in the first example the shorthand for “M_N_I_long_t” would not be allowed, since it clashes with “M_long_t” and “M_N_long_t”. If these shorter forms are not required, they can be ignored by setting

#define SOM_DONT_USE_SHORT_NAMES before including the public header files, or by using the SOM Compiler option –mnouseshort so that they are not generated in the header files.

In the SOM documentation and samples, both long and short forms are illustrated, for both type names and method calls. It is the responsibility of each user to adopt a style according to personal preference. It should be noted, however, that CORBA specifies that only the long forms must be present.

Union type

IDL also supports a **union** type, which is a cross between the C *union* and *switch* statements. The syntax of a **union** type declaration is as follows:

union *identifier* **switch (** *switch–type* **)**
    { *case+* **}**

The "identifier" following the **union** keyword defines a new legal type. (**Union** types may also be named using a **typedef** declaration.) The "switch–type" specifies an integral, character, boolean, or enumeration type, or the name of a previously defined integral, boolean, character, or enumeration type. Each "case" of the **union** is specified with the following syntax:

case–label+   type–spec   declarator ;

where "type-spec" is any valid type specification; "declarator" is an identifier, an array declarator (such as, foo[3][5]), or a pointer declarator (such as, *foo); and each "case-label" has one of the following forms:

case *const–expr***:**
default:

The "const-expr" is a constant expression that must match or be automatically castable to the "switch-type". A **default** case can appear no more than once.

Unions are mapped onto C/C++ **structs**. For example, the following IDL declaration:

```
union Foo switch (long) {
  case 1: long x;
  case 2: float y;
  default: char z;
};
```

is mapped onto the following C struct:

```
typedef Hello_struct {
  long _d;
  union {
    long x;
    float y;
    char z;
  } _u;
} Hello_foo;
```

The discriminator is referred to as "_d", and the union in the struct is referred to as "_u". Hence, elements of the union are referenced just as in C:

```
Foo v;

/* get a pointer to Foo in v: */
switch(v->_d) {
  case 1: printf("x = %ld\n", v->_u.x); break;
  case 2: printf("y = %f\n", v->_u.y); break;
  default: printf("z = %c\n", v->_u.z); break;
}
```

Note: This example is from *The Common Object Request Broker: Architecture and Specification,* revision 1.1, page 90.

Template types (sequences and strings)

IDL defines two template types not found in C and C++: **sequences** and **strings**. A **sequence** is a one-dimensional array with two characteristics: a maximum size (specified at compile time) and a length (determined at run time). **Sequences** permit passing unbounded arrays between objects. **Sequences** are specified as follows:

sequence < *simple–type* [**,** *positive–integer–const*] **>**

where “simple-type” specifies any valid IDL type, and the optional “positive-integer-const” is a constant expression that specifies the maximum size of the **sequence** (as a positive integer).

Note: The “simple–type” cannot have a ‘*’ directly in the sequence statement. Instead, a typedef for the pointer type must be used. For example, instead of:

```
typedef sequence<long *> seq_longptr;   // Error: '*' not allowed.
```

use:

```
typedef long * longptr;
typedef sequence<longptr> seq_longptr;  // Ok.
```

In SOM’s C and C++ bindings, **sequences** are mapped onto **structs** with the following members:

unsigned long _maximum;
unsigned long _length;
simple–type ***_buffer;**

where “simple-type” is the specified type of the **sequence**. For example, the IDL declaration

```
typedef sequence<long, 10> vec10;
```

results in the following C **struct**:

```
#ifndef _IDL_SEQUENCE_long_defined
#define _IDL_SEQUENCE_long_defined
typedef struct {
    unsigned long _maximum;
    unsigned long _length;
    long *_buffer;
} _IDL_SEQUENCE_long;
#endif /* _IDL_SEQUENCE_long_defined */
typedef _IDL_SEQUENCE_long vec10;
```

and an instance of this type is declared as follows:

```
vec10 v = {10L, 0L, (long *)NULL};
```

The “_maximum” member designates the actual size of storage allocated for the **sequence**, and the “_length” member designates the number of values contained in the “_buffer” member. For bounded **sequences**, it is an error to set the “_length” or “_maximum” member to a value larger than the specified bound of the **sequence**.

Before a **sequence** is passed as the value of an “in” or “inout” method parameter, the “_buffer” member must point to an array of elements of the appropriate type, and the “_length” member must contain the number of elements to be passed. (If the parameter is “inout” and the **sequence** is unbounded, the “_maximum” member must also be set to the actual size of the array. Upon return, “_length” will contain the number of values copied into “_buffer”, which must be less than “_maximum”.) When a **sequence** is passed as an “out” method parameter or received as the return value, the method procedure allocates storage for “_buffer” as needed, the “_length” member contains the number of elements returned, and the “_maximum” member contains the number of elements allocated. (The client is responsible for subsequently freeing the memory pointed to by “_buffer”.)

C and C++ programs using SOM's language bindings can refer to **sequence** types as:

_IDL_SEQUENCE_*type*

where "type" is the effective type of the **sequence** members. For example, the IDL type `sequence<long,10>` is referred to in a C/C++ program by the type name `_IDL_SEQUENCE_long`. If `longint` is defined via a typedef to be type `long`, then the IDL type `sequence<longint,10>` is also referred to by the type name `_IDL_SEQUENCE_long`.

If the typedef is for a pointer type, then the effective type is the name of the pointer type. For example, the following statements define a C/C++ type `_IDL_SEQUENCE_longptr` and *not* `_IDL_SEQUENCE_long`:

```
typedef long * longptr;
typedef sequence<longptr> seq_longptr;
```

A **string** is similar to a **sequence** of type **char.** It can contain all possible 8-bit quantities except NULL. **Strings** are specified as follows:

string [< *positive–integer–const* > **]**

where the optional "positive-integer-const" is a constant expression that specifies the maximum size of the **string** (as a positive integer, which does not include the extra byte to hold a NULL as required in C/C++). In SOM's C and C++ bindings, **strings** are mapped onto zero-byte terminated character arrays. The length of the string is encoded by the position of the zero-byte. For example, the following IDL declaration:

```
typedef string<10> foo;
```

is converted to the following C/C++ **typedef**:

```
typedef char *foo;
```

A variable of this type is then declared as follows:

```
foo s = (char *) NULL;
```

C and C++ programs using SOM's language bindings can refer to **string** types by the type name *string*.

When an unbounded **string** is passed as the value of an "inout" method parameter, the returned value is constrained to be no longer than the input value. Hence, using unbounded **string**s as "inout" parameters is not advised.

Arrays

Multidimensional, fixed-size arrays can be declared in IDL as follows:

identifier { **[** *positive–integer–const* **]** }+

where the "positive-integer-const" is a constant expression that specifies the array size, in each dimension, as a positive integer. The array size is fixed at compile time.

Pointers

Although the CORBA standard for IDL does not include them, SOM IDL offers pointer types. Declarators of a pointer type are specified as in C and C++:

*type *declarator*

where "type" is a valid IDL type specification and "declarator" is an identifier or an array declarator.

Object types

The name of the interface to a class of objects can be used as a type. For example, if an IDL specification includes an **interface** declaration (described below) for a class (of objects) "C1", then "C1" can be used as a type name within that IDL specification. When used as a type, an interface name indicates a pointer to an object that supports that interface. An interface name can be used as the type of a method argument, as a method return type, or as the type of a member of a constructed type (a **struct, union**, or **enum**). In all cases, the use of an interface name implicitly indicates a pointer to an object that supports that interface.

As explained in Chapter 3, SOM's C usage bindings for SOM classes also follow this convention. However, within SOM's C++ bindings, the pointer is made explicit, and the use of an interface name as a type refers to a class instance itself, rather than a pointer to a class instance. For example, to declare a variable "myobj" that is a pointer to an instance of class "Foo" in an IDL specification and in a C program, the following declaration is required:

```
Foo myobj;
```

However, in a C++ program, the following declaration is required:

```
Foo *myobj;
```

If a C programmer uses the SOM Compiler option **–maddstar,** then the bindings generated for C will also require an explicit '*' in declarations. Thus,

`Foo myobj;`	in IDL requires
`Foo *myobj;`	in both C and C++ programs.

This style of bindings for C is permitted for two reasons:

- It more closely resembles the bindings for C++, thus making it easier to change to the C++ bindings at a later date; and
- It is required for compatibility with existing SOM OIDL code.

Note: The same C and C++ binding emitters should *not* be run in the same SOM Compiler command. For example,

```
sc "-sh;xh" cls.idl    // Not valid.
```

If you wish to generate both C and C++ bindings, you should issue the commands separately:

```
sc -sh cls.idl
sc -sxh cls.idl
```

Exception declarations

IDL specifications may include **exception** declarations, which define data structures to be returned when an exception occurs during the execution of a method. (IDL exceptions are implemented by simply passing back error information after a method call, as opposed to the "catch/throw" model where an exception is implemented by a long jump or signal.) Associated with each type of exception is a name and, optionally, a struct-like data structure for holding error information. Exceptions are declared as follows:

exception *identifier* **{** *member** **};**

The "identifier" is the name of the exception, and each "member" has the following form:

type–spec declarators ;

where "type-spec" is a valid IDL type specification and "declarators" is a list of identifiers, array declarators, or pointer declarators, delimited by commas. The members of an exception structure should contain information to help the caller understand the nature of the error. The exception declaration can be treated like a **struct** definition; that is, whatever you can access in an IDL **struct**, you can access in an **exception** declaration. Alternatively, the structure can be *empty*, whereby the exception is just identified by its name.

If an **exception** is returned as the outcome of a method, the exception "identifier" indicates which exception occurred. The values of the members of the exception provide additional information specific to the exception. The topic "Method declarations" below describes how to indicate that a particular method may raise a particular exception, and Chapter 3, "Using SOM Classes in Client Programs," describes how exceptions are handled, in the section entitled "Exceptions and error handling."

Following is an example declaration of a "BAD_FLAG" exception:

```
exception BAD_FLAG { long ErrCode; char Reason[80]; };
```

The SOM Compiler will map the above exception declaration to the following C language constructs:

```
#define ex_BAD_FLAG "::BAD_FLAG"
typedef struct BAD_FLAG {
    long  ErrCode;
    char  Reason[80];
} BAD_FLAG;
```

Thus, the `ex_BAD_FLAG` symbol can be used as a shorthand for naming the exception.

An exception declaration within an interface "Hello", such as this:

```
interface Hello {
    exception LOCAL_EXCEPTION { long ErrCode; };
};
```

would map onto:

```
#define ex_Hello_LOCAL_EXCEPTION "::Hello::LOCAL_EXCEPTION"
typedef struct Hello_LOCAL_EXCEPTION {
    long  ErrCode;
} Hello_LOCAL_EXCEPTION;
#define ex_LOCAL_EXCEPTION ex_Hello_LOCAL_EXCEPTION
```

In addition to user-defined exceptions, there are several predefined exceptions for system run-time errors. The standard exceptions as prescribed by CORBA are shown in Table 2. These exceptions correspond to standard run-time errors that may occur during the execution of any method (regardless of the list of exceptions listed in its IDL specification).

Each of the standard exceptions has the same structure: an error code (to designate the subcategory of the exception) and a completion status code. For example, the `NO_MEMORY` standard exception has the following definition:

```
enum completion_status {YES, NO, MAYBE};
exception NO_MEMORY { unsigned long minor;
                      completion_status completed; };
```

The "completion_status" value indicates whether the method was never initiated (NO), completed its execution prior to the exception (YES), or the completion status is indeterminate (MAYBE).

Since all the standard exceptions have the same structure, **somcorba.h** (included by **som.h**) defines a generic **StExcep** typedef which can be used instead of the specific typedefs:

```
typedef struct StExcep {
     unsigned long minor;
     completion_status completed;
} StExcep;
```

The standard exceptions shown in Table 2 are defined in an IDL module called **StExcep**, in the file called **stexcep.idl**, and the C definitions can be found in **stexcep.h**.

Table 2. STANDARD EXCEPTIONS DEFINED BY CORBA

```
module StExcep {
    #define ex_body { unsigned long minor; completion_status completed; }

    enum completion_status { YES, NO, MAYBE };
    enum exception_type {NO_EXCEPTION, USER_EXCEPTION, SYSTEM_EXCEPTION};

    exception UNKNOWN           ex_body;  // the unknown exception
    exception BAD_PARAM         ex_body;  // an invalid parameter was passed
    exception NO_MEMORY         ex_body;  // dynamic memory allocation failure
    exception IMP_LIMIT         ex_body;  // violated implementation limit
    exception COMM_FAILURE      ex_body;  // communication failure
    exception INV_OBJREF        ex_body;  // invalid object reference
    exception NO_PERMISSION     ex_body;  // no permission for attempted op.
    exception INTERNAL          ex_body;  // ORB internal error
    exception MARSHAL           ex_body;  // error marshalling param/result
    exception INITIALIZE        ex_body;  // ORB initialization failure
    exception NO_IMPLEMENT      ex_body;  // op. implementation unavailable
    exception BAD_TYPECODE      ex_body;  // bad typecode
    exception BAD_OPERATION     ex_body;  // invalid operation
    exception NO_RESOURCES      ex_body;  // insufficient resources for request
    exception NO_RESPONSE       ex_body;  // response to req. not yet available
    exception PERSIST_STORE     ex_body;  // persistent storage failure
    exception BAD_INV_ORDER     ex_body;  // routine invocations out of order
    exception TRANSIENT         ex_body;  // transient failure – reissue request
    exception FREE_MEM          ex_body;  // cannot free memory
    exception INV_IDENT         ex_body;  // invalid identifier syntax
    exception INV_FLAG          ex_body;  // invalid flag was specified
    exception INTF_REPOS        ex_body;  // error accessing interface repository
    exception CONTEXT           ex_body;  // error processing context object
    exception OBJ_ADAPTER       ex_body;  // failure detected by object adapter
    exception DATA_CONVERSION   ex_body;  // data conversion error

};
```

Interface declarations

The IDL specification for a class of objects must contain a declaration of the **interface** these objects will support. Because, in SOM, objects are implemented using classes, the interface name is always used as a class name as well. Therefore, an interface declaration can be understood to specify a class name, and its parent (direct base) class names. This is the approach used in the following description of an interface declaration. In addition to the class name and its parents names, an interface indicates new methods (operations), and any constants, type definitions, and exception structures that the interface exports. An interface declaration has the following syntax:

interface class–name [**:** parent–class1**,** parent–class2, ...]
{
constant declarations (optional)
type declarations (optional)
exception declarations (optional)
attribute declarations (optional)
method declarations (optional)
implementation statement (optional)
};

Many class implementors distinguish a "class-name" by using an initial capital letter, but that is optional. The "parent-class" (or base-class) names specify the interfaces from which the interface of "class-name" instances is derived. Parent-class names are required only for the immediate parent(s). Each parent class must have its own IDL specification (which must be *#included* in the subclass's .idl file). A parent class cannot be named more than once in the **interface** statement header.

Note: In general, an "**interface** *<className>*" header must precede any subsequent implementation that references *<className>*. For more discussion of multiple **interface** statements, refer to the later topic "Defining multiple interfaces in a .idl file."

The following topics describe the various declarations/statements that can be specified within the body of an **interface** declaration. The order in which these declarations are specified is usually optional, and declarations of different kinds can be intermixed. Although all of the declarations/statements are listed above as "optional," in some cases using one of them may mandate another. For example, if a **method** raises an **exception**, the exception structure must be defined beforehand. In general, **types, constants,** and **exceptions,** as well as **interface** declarations, must be defined before they are referenced, as in C/C++.

Constant, type, and exception declarations

The form of a **constant, type**, or **exception** declaration within the body of an **interface** declaration is the same as described previously in this chapter. **Constants** and **types** defined <u>within</u> an **interface** for a class are transferred by the SOM Compiler to the binding files it generates for that class, whereas **constants** and **types** defined <u>outside</u> of an **interface** are not.

Global types (such as, those defined outside of an interface and module) can be emitted by surrounding them with the following **#pragmas**:

```
#pragma somemittypes on
    typedef sequence <long,10> vec10;
    exception BAD_FLAG { long ErrCode; char Reason[80]; };
    typedef long long_t;
#pragma somemittypes off
```

Types, constants, and **exceptions** defined in a parent class are also accessible to the child class. References to them, however, must be unambiguous. Potential ambiguities can be resolved by prefacing a name with the name of the class that defines it, separated by the characters "::" as illustrated below:

```
MyParentClass::myType
```

The child class can redefine any of the **type, constant,** and **exception** names that have been inherited, although this is not advised. The derived class cannot, however, redefine **attributes** or **methods**. It can only replace the implementation of **methods** through overriding (as in example 3 of the Tutorial). To refer to a **constant, type,** or **exception** "name" defined by a parent class and redefined by "class-name," use the "parent-name::name" syntax as before.

Note: A name reference such as `MyParentClass::myType` required in IDL syntax is equivalent to `MyParentClass_myType` in C/C++. For a full discussion of name recognition in SOM, see "Scoping and name resolution" later in this chapter.

Attribute declarations

Declaring an **attribute** as part of an **interface** is equivalent to declaring two accessor methods: one to retrieve the value of the **attribute** (a "get" method, named "_get_<attributeName>") and one to set the value of the **attribute** (a "set" method, named "_set_<attributeName>").

Attributes are declared as follows:

[**readonly**] **attribute** *type–spec* *declarators* **;**

where "type-spec" specifies any valid IDL type and "declarators" is a list of identifiers or pointer declarators, delimited by commas. (An array declarator cannot be used directly when declaring an **attribute**, but the type of an attribute can be a user-defined type that is an array.) The optional **readonly** keyword specifies that the value of the **attribute** can be accessed but not modified by client programs. (In other words, a **readonly attribute** has no "set" method.) Below are examples of **attribute** declarations, which are specified within the body of an **interface** statement for a class:

```
interface Goodbye: Hello, SOMObject
{
  void  sayBye();

  attribute short xpos;
  attribute char c1, c2;
  readonly attribute float xyz;
};
```

The preceding **attribute** declarations are equivalent to defining the following methods:

```
short _get_xpos();
void  _set_xpos(in short xpos);
char  _get_c1();
void  _set_c1(in char c1);
char  _get_c2();
void  _set_c2(in char c2);
float _get_xyz();
```

Note: Although the preceding attribute declarations are equivalent to the explicit method declarations above, these method declarations are *not* legal IDL, because the method names begin with an '_'. All IDL identifiers must begin with an alphabetic character, not including '_'.

Attributes are inherited from ancestor classes (indirect base classes). An inherited **attribute** name cannot be redefined to be a different type.

Method (operation) declarations

Method (operation) declarations define the interface of each method introduced by the class. A method declaration is similar to a C/C++ function definition:

[**oneway**] *type–spec* *identifier* **(** *parameter–list* **)** [*raises–expr*] [*context–expr*] **;**

where "identifier" is the name of the method and "type-spec" is any valid IDL **type** (or the keyword **void**, indicating that the method returns no value). Unlike C and C++ procedures, methods that do not return a result must specify **void** as their return type. The remaining syntax of a method declaration is elaborated in the following subtopics.

Note: Although IDL does not allow methods to receive and return values whose type is a pointer to a function, it does allow methods to receive and return method names (as **string** values). Thus, rather than defining methods that pass pointers to functions (and that subsequently invoke those functions), programmers should instead define methods that pass method names (and subsequently invoke those methods using one of the SOM-supplied method-dispatching or method-resolution methods or functions, such as **somDispatch**).

Oneway keyword

The optional **oneway** keyword specifies that when a client invokes the method, the invocation semantics are "best-effort", which does not guarantee delivery of the call. "Best-effort" implies that the method will be invoked at most once. A **oneway** method should not have any output parameters and should have a return type of **void**. A **oneway** method also should not include a "raises expression" (see below), although it may raise a standard exception.

If the **oneway** keyword is not specified, then the method has "at-most-once" invocation semantics if an exception is raised, and it has "exactly-once" semantics if the method succeeds. This means that a method that raises an exception has been executed zero or one times, and a method that succeeds has been executed exactly once.

Note: Currently the "oneway" keyword, although accepted, has no effect on the C/C++ bindings that are generated.

Parameter list

The "parameter-list" contains zero or more parameter declarations for the method, delimited by commas. (The target object for the method is not explicitly specified as a method parameter in IDL, nor are the **Environment** or **Context** parameters.) If there are no explicit parameters, the syntax "( )" must be used, rather than "(void)". A parameter declaration has the following syntax:

{ **in** | **out** | **inout** } *type–spec declarator*

where "type-spec" is any valid IDL type and "declarator" is an identifier, array declarator, or pointer declarator.

<u>**In, out, inout** parameters</u>: The required **in**|**out**|**inout** directional attribute indicates whether the parameter is to be passed from client to server (**in**), from server to client (**out**), or in both directions (**inout**). A method must not modify an **in** parameter. If a method raises an exception, the values of the return result and the values of the **out** and **inout** parameters (if any) are undefined. When an unbounded **string** or **sequence** is passed as an **inout** parameter, the returned value must be no longer than the input value.

The following are examples of valid method declarations in SOM IDL:

```
short meth1(in char c, out float f);
oneway void meth2(in char c);
float meth3();
```

Classes derived from **SOMObject** can declare methods that take a pointer to a block of memory containing a variable number of arguments, using a final parameter of type **va_list**. The **va_list** must use the parameter name "ap", as in the following example:

```
void MyMethod(in short numArgs, in va_list ap);
```

For **in** parameters of type **array**, C and C++ clients must pass the address of the first element of the array. For **in** parameters of type **struct, union, sequence** or **any**, C/C++ clients must pass the address of a variable of that type, rather than the variable itself.

For all IDL types except **arrays**, if a parameter of a method is **out** or **inout**, then C/C++ clients must pass the address of a variable of that type (or the value of a pointer to that variable) rather than the variable itself. (For example, to invoke method "meth1" above, a pointer to a variable of type **float** must be passed in place of parameter "f".) For **arrays**, C/C++ clients must pass the address of the first element of the **array**.

If the return type of a method is a **struct, union, sequence**, or **any** type, then for C/C++ clients, the method returns the value of the C/C++ struct representing the IDL **struct, union, sequence**, or **any**. If the return type is **string,** then the method returns a pointer to the first character of the **string**. If the return type is **array**, then the method returns a pointer to the first element of the **array**.

The pointers implicit in the parameter types and return types for IDL method declarations are made explicit in SOM's C and C++ bindings. Thus, the stub procedure that the SOM Compiler generates for method “meth1”, above, has the following signature:

```
SOM_Scope short  SOMLINK meth1(char c, float *f)
```

For C and C++ clients, if a method has an **out** parameter of type **string, sequence,** or **any**, then the method must allocate the storage for the **string,** for the “_buffer” member of the struct that represents the **sequence,** or for the “_value” member of the struct that represents the **any**. It is then the responsibility of the client program to free the storage when it is no longer needed. Similarly, if the return type of a method is **string**, **sequence**, **any**, or **array**, then storage must be allocated by the method, and the client program is responsible for subsequently freeing it.

Note: The foregoing description also applies for the **_get_**<*attributeName*> method associated with an attribute of type **string**, **sequence**, **any**, or **array**. Hence, the attribute should be specified with a “noget” modifier to override automatic implementation of the attribute's “get” method. Then, needed memory can be allocated by the developer's “get” method implementation and subsequently deallocated by the caller. (The “noget” modifier is described under the topic “Modifier statements” later in this section.)

Raises expression

The optional **raises** expression (“raises-expr”) in a method declaration indicates which exceptions the method may raise. (IDL exceptions are implemented by simply passing back error information after a method call, as opposed to the “catch/throw” model where an exception is implemented by a long jump or signal.) A **raises** expression is specified as follows:

raises (*identifier1***,** *identifier2***,** ... **)**

where each “identifier” is the name of a previously defined **exception**. In addition to the exceptions listed in the **raises** expression, a method may also signal any of the standard exceptions. Standard exceptions, however, should not appear in a **raises** expression. If no **raises** expression is given, then a method can raise only the standard exceptions. (See the earlier topic “Exception declarations” for information on defining exceptions and for the list of standard exceptions. See Chapter 3, the section entitled “Exceptions and error handling,” for information on using exceptions.)

Context expression

The optional **context expression** (“context-expr”) in a method declaration indicates which elements of the client's context the method may consult. A context expression is specified as follows:

context (*identifier1***,** *identifier2***,** ... **)**

where each “identifier” is a string literal made up of alphanumeric characters, periods, underscores, and asterisks. (The first character must be alphabetic, and an asterisk can only appear as the last character, where it serves as a wildcard matching any characters. If convenient, identifiers may consist of period-separated valid identifier names, but that form is optional.)

The **Context** is a special object that is specified by the CORBA standard. It contains a property list — a set of property-name/string-value pairs that the client can use to store information about its environment that methods may find useful. It is used in much the same way as environment variables. It is passed as an additional (third) parameter to CORBA-compliant methods that are defined as “context-sensitive” in IDL, along with the CORBA-defined Environment structure.

The **context expression** of a method declaration in IDL specifies which property names the method uses. If these properties are present in the **Context** object supplied by the client, they will be passed to the object implementation, which can access them via the **get_values** method of the **Context** object. However, the argument that is passed to the method having a context expression is a **Context** object, *not* the names of the properties. The client program must either create a **Context** object and use the **set_values** or **set_one_value** method of the **Context** class to set the context properties, or use the **get_default_context** method. The client program then passes the **Context** object in the method invocation. Note that the CORBA standard also allows properties in addition to those in the **context** expression to be passed in the **Context** object.

In Chapter 3, "Using SOM Classes in Client Programs," the topic "Invoking Methods" describes the placement of a **context** parameter in a method call. See also chapter 6 of *The Common Object Request Broker: Architecture and Specification* for a discussion of how clients associate values with **context** identifiers. A description of the **Context** class and its methods is contained in the *SOMobjects Developer Toolkit: Programmers Reference Manual.*

Implementation statements

A SOM IDL interface statement for a class may contain an **implementation** statement, which specifies information about how the class will be implemented (version numbers for the class, overriding of inherited methods, what resolution mechanisms the bindings for a particular method will support, and so forth). If the **implementation** statement is omitted, default information is assumed.

Because the **implementation** statement is specific to SOM IDL (and is not part of the CORBA standard), the **implementation** statement should be preceded by an "#ifdef _ _SOMIDL_ _" directive and followed by an "#endif" directive. (See Example 3 in the SOM IDL Tutorial presented earlier.)

The syntax for the implementation statement is as follows:

#ifdef __SOMIDL__
implementation
{
  *implementation**
};
#endif

where each "implementation" can be a **modifier** statement, a **passthru** statement, or a declarator of an **instance variable**, terminated by a semicolon. These constructs are described below. An **interface** statement may *not* contain multiple **implementation** statements.

Modifier statements

A **modifier** statement gives additional implementation information about IDL definitions, such as **interfaces**, **attributes**, **methods**, and **types**. Modifiers can be unqualified or qualified: An **unqualified modifier** is <u>associated with the interface</u> it is defined in. An unqualified modifier statement has the following two syntactic forms:

modifier
modifier **=** *value*

where "modifier" is either a SOM Compiler-defined identifier or a user-defined identifier, and where "value" is an identifier, a string enclosed in double quotes (" "), or a number.

For example:

```
filestem = foo;
nodata;
persistent;
dllname = "E:/som/dlls";
```

A **qualified modifier** is associated with a qualifier. The qualified modifier has the syntax:

qualifier **:** *modifier*
qualifier **:** *modifier* **=** *value*
#pragma modifier *qualifier* **:** *modifier*
#pragma modifier *qualifier* **:** *modifier* **=** *value*

where “qualifier” is the identifier of an IDL definition or is user defined. If the “qualifier” is an IDL definition introduced in the current interface, module, or global scope, then the modifier is attached to that definition. Otherwise, if the qualifier is user defined, the modifier is attached to the interface it occurs in. If a user-defined modifier is defined outside of an interface body (by using **#pragma modifier**), then it is ignored.

For example, consider the following IDL file. (Notice that qualified modifiers can be defined with the “qualifier” and “modifier[=value]” in either order. Also observe that additional modifiers can be included by separating them with commas.)

```
#include <somobj.idl>
#include <somcls.idl>


typedef long newInt;
#pragma somemittypes on
#pragma modifier newInt : nonportable;
#pragma somemittypes off
module M {
    typedef long long_t;
    module  N {
        typedef short short_t;
        interface M_I : SOMClass {
            implementation {
                somInit : override;
            };
        };
        interface I : SOMObject {
            void op ();
            #pragma modifier op : persistent;


            typedef char char_t;
            implementation {
                releaseorder : op;
                metaclass = M_I;
                callstyle = oidl;
                mymod : a, b;
                mymod : c, d;
                e     : mymod;
                f     : mymod;
                op : persistent;
            };
        };
    };
};
```

From the preceding IDL file, modifiers are associated with the following definitions:

```
TypeDef "::newInt"              1  modifier: nonportable
InterfaceDef "::M::N::M_I"      1  modifier: override = somInit
InterfaceDef "::M::N::I"        9 modifiers: metaclass = M_I,
                                             releaseorder = op
                                             callstyle = oidl
                                             mymod = a,b,c,d,e,f
                                             a = mymod
                                             b = mymod
                                             c = mymod
                                             d = mymod
                                             e = mymod
                                             f = mymod
OperationDef "::M::N::I::op"    1  modifier: persistent
```

Notice how the modifiers for the user-defined qualifier "mymod":

```
mymod : a, b;
mymod : c, d;
e     : mymod;
f     : mymod;
```

map onto:

```
mymod = a,b,c,d,e,f
a     = mymod
b     = mymod
c     = mymod
d     = mymod
e     = mymod
f     = mymod
```

This enables users to look up the modifiers with "mymod", either by looking for "mymod" or by using each individual value that uses "mymod". These user-defined modifiers are available for Emitter writers (see the *Emitter Framework Guide and Reference*) and from the Interface Repository (see Chapter 7, "The Interface Repository Framework").

SOM Compiler unqualified modifiers

Unqualified modifiers (described below) include the SOM Compiler-defined identifiers: **abstract, baseproxyclass, callstyle, classinit, directinitclasses, dllname, filestem, functionprefix, majorversion, metaclass, memory_management, minorversion, somallocate,** and **somdeallocate.**

abstract — Specifies that the class is intended for use as a parent for subclass derivations, but not for creating instances.

baseproxyclass = *class* — Specifies the base proxy class to be used by DSOM when dynamically creating a proxy class for the current class. The base proxy class must be derived from the class **SOMDClientProxy.** The **SOMDClientProxy** class will be used if the **baseproxyclass** modifier is unspecified. (See Chapter 6, "Distributed SOM," for a discussion on customizing proxy classes.)

callstyle = oidl — Specifies that the method stub procedures generated by SOM's C/C++ bindings will not include the CORBA-specified (*Environment *ev)* and (*context *ctx*) parameters.

classinit = *procedure* — Specifies a user-written procedure that will be executed to initialize the class object when it is created. If the **classinit** modifier is specified in the .idl file for a class, the SOM Compiler will provide a template for the procedure in the implementation file it generates. The class implementor can then fill in the body of this procedure template.

directinitclasses = "*ancestor1, ancestor2, ...*" — Specifies the ancestor class(es) whose initializers (and destructors) will be directly invoked by this class's initialization (and destruction) routines. If this modifier is not explicitly specified, the default setting is the parents of the class. For further information, see "Initializing and Uninitializing Objects" in Chapter 5, "Implementing Classes in SOM."

dllname = *filename* — Specifies the name of the library file that will contain the class's implementation. If *filename* contains special characters (e.g., periods, backslashes), then *filename* should be surrounded by double quotes (""). The *filename* specified can be either a full pathname, or an unqualified (or partially qualified) filename. In the latter cases, the environment variable LIBPATH (on AIX or OS/2) or PATH (on Windows) is used to locate the file.

filestem = *stem* — Specifies how the SOM Compiler will construct file names for the binding files it generates (<stem>.h, <stem>.c, etc.). The default stem is the file stem of the .idl file for the class.

functionprefix = *prefix* — Directs the SOM Compiler to construct method-procedure names by prefixing method names with "prefix". For example, "functionprefix = xx;" within an **implementation** statement would result in a procedure name of `xxfoo` for method `foo`. The default for this attribute is the empty string. If an interface is defined in a module, then the default function prefix is the fully scoped interface name. *Tip*: Using a function prefix with the same name as the class makes it easier to remember method-procedure names when debugging.

When an .idl file defines multiple interfaces not contained within a module, use of a function prefix for each interface is essential to avoid name collisions. For example, if one interface introduces a method and another interface in the same .idl file overrides it, then the implementation file for the classes will contain two method procedures of the same name (unless function prefixes are defined for one of the classes), resulting in a name collision at compile time.

majorversion = *number* — Specifies the major version number of the current class definition. The major version number of a class definition usually changes only when a significant enhancement or incompatible change is made to the class. The "number" must be a positive integer less than 2^{32}–1. If a non-zero major version number is specified, SOM will verify that any code that purports to implement the class has the same major version number. The default major version number is zero.

memory_management = corba

— Specifies that all methods introduced by the class follow the CORBA specification for parameter memory management, except where a particular method has an explicit modifier indicating otherwise (either "object_owns_result" or "object_owns_parameters"). See the section in Chapter 6 entitled "Memory Management" for a discussion of the CORBA memory-management requirements.

metaclass = *class* — Specifies the class's metaclass. The specified metaclass (or one automatically derived from it at run time) will be used to create the class object for the class. If a **metaclass** is specified, its .idl file (if separate) must be included in the **include** section of the class's .idl file. If no metaclass is specified, the metaclass will be defined automatically.

minorversion = *number* — Specifies the minor version number of the current class definition. The minor version number of a class definition changes whenever minor enhancements or fixes are made to a class. Class implementors usually maintain backward compatibility across changes in the minor version number. The "number" must be a positive integer less than 2^{32}–1. If a non-zero minor version number is specified, SOM will verify that any code that purports to implement the class has the same or a higher minor version number. The default minor version number is zero.

somallocate = *procedure* — Specifies a user-written procedure that will be executed to allocate memory for class instances when the **somAllocate** class method is invoked.

somdeallocate = *procedure*

— Specifies a user-written procedure that will be executed to deallocate memory for class instances when the **somDeallocate** class method is invoked.

The following example illustrates the specification of unqualified interface modifiers:

```
implementation
{
  filestem = hello;
  functionprefix = hel;
  majorversion = 1;
  minorversion = 2;
  classinit = helloInit;
  metaclass = M_Hello;
};
```

SOM Compiler qualified modifiers

Qualified modifiers are categorized according to the IDL component (class, attribute, method, or type) to which each modifier applies. Listed below are the SOM Compiler-defined identifiers used as qualified modifiers, along with the IDL component to which it applies. Descriptions of all qualified modifiers are then given in alphabetical order. Recall that qualified modifiers are defined using the syntax *qualifier* : *modifier*[**=***value*].

For classes:
  releaseorder

For attributes:
  indirect, nodata, noget, noset, and **persistent**

For methods:

caller_owns_parameters, caller_owns_result, const, init, method, migrate, namelookup, nocall, noenv, nonstatic, nooverride, noself, object_owns_parameters, object_owns_result, offset, override, procedure, reintroduce, and **select**

For variables:

staticdata

For types:

impctx

The following paragraphs describe each qualified modifier.

caller_owns_parameters = "*p1, p2, ..., pn"* — Specifies the names of the method's parameters whose ownership is retained by (in the case of "in" parameters) or transferred to (for "inout" or "out" parameters) the caller. This modifier is only valid in the interface specification of the method's introducing class. This modifier only makes sense for parameters whose IDL type is a data item that can be freed (string, object, array, pointer, or TypeCode), or a data item containing memory that can be freed (for example, a sequence or any), or a struct or union.

For parameters whose type is an object, ownership applies to the object *reference* rather than to the object (that is, the caller should invoke **release** on the parameter, rather than **somFree**).

caller_owns_result — Specifies that ownership of the return result of the method is transferred to the caller, and that the caller is responsible for freeing the memory. This modifier is only valid in the interface specification of the method's introducing class. This modifier only makes sense when the method's return type is a data type that can be freed (string, object, array, pointer, or TypeCode), or a data item containing memory that can be freed (for example, a sequence or any). For methods that return an object, ownership applies to the object reference rather than to the object (that is, the caller should invoke **release** on the result, rather than **somFree**).

const — Indicates that implementations of the related method should not modify their target argument. SOM provides no way to verify or guarantee that implementations do not modify the targets of such methods, and the information provided by this modifier is not currently of importance to any of the Toolkit emitters. However, the information may prove useful in the future. For example, since modifiers are available in the Interface Repository, there may be future uses of this information by DSOM.

impctx — Supports types that cannot be fully defined using IDL. For full information, see "Using the tk_foreign TypeCode" in Chapter 7, "The Interface Repository Framework."

indirect — Directs the SOM Compiler to generate "get" and "set" methods for the attribute that take and return a pointer to the attribute's value, rather than the attribute value itself. For example, if an attribute x of type float is declared to be an indirect attribute, then the "_get_x" method will return a pointer to a float, and the input to the "_set_x" method must be a pointer to a float. (This modifier is provided for OIDL compatibility only.)

init — Indicates that a method is an initializer method. For information concerning the use of this modifier, see "Initializing and Uninitializing Objects" in Chapter 5, "Implementing Classes in SOM."

method or **nonstatic** or **procedure** — Indicates the category of method implementation. Refer to the topic "The four kinds of SOM methods" in Chapter 5, "Implementing Classes in SOM," for an explanation of the meanings of these different method modifiers. If none of these modifiers is specified, the default is **method**. Methods with the **procedure** modifier cannot be invoked remotely using DSOM.

migrate = *ancestor* — Indicates that a method originally introduced by this interface has been moved upward to a specified <*ancestor*> interface. When this is done, the method introduction must be removed from this interface (because the method is now inherited). However, the original **releaseorder** entry for the method should be retained, and **migrate** should be used to assure that clients compiled based on the original interface will not require recompilation. The ancestor interface is specified using a C–scoped interface name. For example, "Module_InterfaceName", not "Module::InterfaceName". See the later topic "Name usage in client programs" for an explanation of C-scoped names.

namelookup — See "**offset** or **namelookup**."

nocall — Specifies that the related method should not be invoked on an instance of this class even though it is supported by the interface.

nodata — Directs the SOM Compiler *not* to define an instance variable corresponding to the attribute. For example, a "time" attribute would not require an instance variable to maintain its value, because the value can be obtained from the operating system. The "get" and "set" methods for "nodata" attributes must be defined by the class implementor; stub method procedures for them are automatically generated in the implementation template for the class by the SOM Compiler.

noenv — Indicates that a direct-call procedure does not receive an environment as an argument.

noget — Directs the SOM Compiler *not* to automatically generate a "get" method procedure for the attribute in the .ih/.xih binding file for the class. Instead, the "get" method must be implemented by the class implementor. A stub method procedure for the "get" method is automatically generated in the implementation template for the class by the SOM Compiler, to be filled in by the implementor.

nonstatic — See "**method** or **nonstatic** or **procedure**."

nooverride — Indicates that the method should not be overridden by subclasses. The SOM Compiler will generate an error if this method is overridden.

noself — Indicates that a direct-call procedure does not receive a target object as an argument.

noset — Directs the SOM Compiler *not* to automatically generate a "set" method procedure for the attribute in the .ih/.xih binding file for the class. Instead, the "set" method must be implemented by the class implementor. A stub method procedure for the "set" method is automatically generated in the implementation template for the class by the SOM Compiler.

Note: The "set" method procedure that the SOM Compiler generates by default for an attribute in the .h/.xh binding file (when the **noset** modifier is *not* used) does a shallow copy of the value that is passed to the attribute. For some attribute types, including strings and pointers, this may not be appropriate. For instance, the "set" method for an attribute of type **string** should perform a string copy, rather than a shallow copy, if the attribute's value may be needed after the client program has freed the memory occupied by the string. In such situations, the class implementor should specify the **noset** attribute modifier and implement the attribute's "set" method manually, rather than having SOM implement the "set" method automatically.

object_owns_parameters = "*p1, p2, ..., pn"*
— Specifies the names of the method's parameters whose ownership is transferred to (in the case of "in" parameters) or is retained by (for "inout" or "out" parameters) the object. For "in" parameters, the object can free the parameter at any time after receiving it. (Hence, the caller should not reuse the parameter or pass it as any other object-owned parameter in the same method call.) For "inout" and "out" parameters, the object is responsible for freeing the parameter sometime before the object is destroyed. This modifier is only valid in the interface specification of the method's introducing class. This modifier only makes sense for parameters whose IDL type is a data item that can be freed (string, object, array, pointer, or TypeCode), or a data item containing memory that can be freed (for example, a sequence or any), or a struct or union.

For parameters whose type is an object, ownership applies to the object *reference* rather than to the object (that is, the object will invoke **release** on the parameter, rather than **somFree**). For "in" and "out" parameters whose IDL-to-C/C++ mapping introduces a pointer, ownership applies only to the data item itself, and not to the introduced pointer. (For example, even if an "out string" IDL parameter (which becomes a "string *" C/C++ parameter) is designated as "object-owned," the object assumes ownership of the string, but not of the pointer to the string.)

object_owns_result — Specifies that the object retains ownership of the return result of the method, and that the caller must not free the memory. The object is responsible for freeing the memory sometime before the object is destroyed. This modifier is only valid in the interface specification of the method's introducing class. This modifier only makes sense when the method's return type is a data type that can be freed (string, object, array, pointer, or TypeCode), or a data item containing memory that can be freed (for example, a sequence or any). For methods that return an object, ownership applies to the object reference rather than to the object (that is, the object will be responsible for invoking **release** on the result, rather than **somFree**).

offset or **namelookup** — Indicates whether the SOM Compiler should generate bindings for invoking the method using offset resolution or name lookup. **Offset** resolution requires that the **class** of the method's target object be known at compile time. When different methods of the same name are defined by several classes, **namelookup** is a more appropriate technique for method resolution than is offset resolution. (See Chapter 3, the section entitled "Invoking Methods.") The default modifier is **offset.**

override — Indicates that the method is one introduced by an ancestor class and that this class will re-implement the method. See also the related modifier, **select**.

persistent — Indicates a persistent attribute of a persistent object. (See Chapter 8, "Persistence Framework," for a discussion of persistent objects.)

procedure — See "**method** or **nonstatic** or **procedure**."

reintroduce — Indicates that this interface will "hide" a method introduced by some ancestor interface, and will replace it with another implementation. Methods introduced as direct-call procedures or nonstatic methods can be reintroduced. However, static methods (the default implementation category for SOM methods) cannot be reintroduced.

releaseorder: *a, b, c, ...* — Specifies the order in which the SOM Compiler will place the class's methods in the data structures it builds to represent the class. Maintaining a consistent release order for a class allows the implementation of a class to change without requiring client programs to be recompiled.

The release order should contain every method name introduced by the class (private and nonprivate), but should not include any inherited methods, even if they are overridden by the class. The "get" and "set" methods defined automatically for each new attribute (named "_get_<attributeName>" and "_set_<attributeName>") should also be included in the release order list. The order of the names on the list is unimportant except that once a name is on the list and the class has client programs, it should not be reordered or removed, even if the method is no longer supported by the class, or the client programs will require recompilation. New methods should be added only to the end of the list. If a method named on the list is to be moved up in the class hierarchy, its name should remain on the current list, but it should also be added to the release order list for the class that will now introduce it.

If not explicitly specified, the release order will be determined by the SOM Compiler, and a warning will be issued for each missing method. If new methods or attributes are subsequently added to the class, the default release order might change; programs using the class would then require recompilation. Thus, it is advisable to explicitly give a release order.

select = *parent* — Used in conjunction with the **override** modifier, this indicates that an inherited static method will use the implementation inherited from the indicated <*parent*> class. The parent is specified using the C-scoped name. For example, “Module_InterfaceName”, not “Module::InterfaceName”. See the later topic “Name usage in client programs” for an explanation of C-scoped names.

staticdata — Indicates that the declared variable is not stored within objects, but, instead, that the ClassData structure for the implementing class will contain a pointer to the staticdata variable. This is similar in concept to C++ static data members. The staticdata variable must also be included in the **releaseorder**. The class implementor has responsibility for loading the ClassData pointer during class initialization. This can be facilitated by writing a special class initialization function and indicating its name using the **classinit** unqualified modifier. *Note:* attributes can be declared as staticdata. This is an important implementation technique that allows classes to introduce attributes whose backing storage is not inherited by subclasses.

The following example illustrates the specification of qualified modifiers:

```
implementation
{
  releaseorder : op1, op3, op2, op5, op6, x, y;
  op1 : persistent;
  somDefaultInit : override, init;
  op2: reintroduce, procedure;
  op3: reintroduce, nonstatic;
  op4: override, select = ModuleName_parentInterfaceName;
  op5: migrate = ModuleName_ancestorInterfaceName;
  op6: procedure, noself, noenv;
  long x;
  x: staticdata;
  y: staticdata; // y is an attribute
  mymod : a, b;
};
```

Passthru statements

A **passthru** statement (used within the body of an **implementation** statement, described above) allows a class implementor to specify blocks of code (for C/C++ programmers, usually only **#include** directives) that the SOM compiler will pass into the header files it generates.

Passthru statements are included in SOM IDL primarily for backward compatibility with the SOM OIDL language, and their use by C and C++ programmers should be limited to **#include**

directives. C and C++ programmers should use IDL **type** and **constant** declarations rather than **passthru** statements when possible. (Users of other languages, however, may require **passthru** statements for type and constant declarations.)

The contents of the **passthru** lines are ignored by the SOM compiler and can contain anything that needs to placed near the beginning of a header file for a class. Even comments contained in **passthru** lines are processed without modification. The syntax for specifying **passthru** lines is one of the following forms:

passthru *language_suffix* = *literal*+ ;
passthru *language_suffix*_**before** = *literal*+ ;
passthru *language_suffix*_**after** = *literal*+ ;

where "language" specifies the programming language and "suffix" indicates which header files will be affected. The SOM Compiler supports suffixes **h, ih, xh,** and **xih.** For both C and C++, "language" is specified as C.

Each "literal" is a string literal (enclosed in double quotes) to be placed verbatim into the specified header file. [Double quotes within the passthru literal should be preceded by a backslash. No other characters escaped with a backslash will be interpreted, and formatting characters (newlines, tab characters, etc.) are passed through without processing.] The last literal for a passthru statement must not end in a backslash (put a space or other character between a final backslash and the closing double quote).

When either of the first two forms is used, **passthru** lines are placed **before** the **#include** statements in the header file. When the third form is used, **passthru** lines are placed just **after** the **#include** statements in the header file.

For example, the following **passthru** statement

```
implementation
{
  passthru C_h = "#include <foo.h>";
};
```

results in the directive `#include <foo.h>` being placed at the beginning of the .h C binding file that the SOM Compiler generates.

Declaring instance variables and staticdata variables

Declarators are used within the body of an **implementation** statement (described above) to specify the *instance variables* that are introduced by a class, and the *staticdata variables* pointed to by the class's **ClassData** structure. These variables are declared using ANSI C syntax for variable declarations, restricted to valid SOM IDL types (see "Type and constant declarations," above). For example, the following **implementation** statement declares two instance variables, x and y, and a staticdata variable, z, for class "Hello," :

```
implementation
{
  short x;
  long y;
  double z;
  z: staticdata;
};
```

Instance variables are normally intended to be accessed only by the class's methods and *not* by client programs or subclasses' methods. For data to be accessed by client programs or subclass methods, *attributes* should be used instead of instance variables. (Note, however, that declaring an attribute has the effect of also declaring an instance variable of the same name, unless the "nodata" attribute modifier is specified.)

Staticdata variables, by contrast, are publicly available and are associated specifically with their introducing class. They are, however, very different in concept from class variables. *Class* *variables* are really instance variables introduced by a metaclass, and are therefore present in any class that is an instance of the introducing metaclass (or of any metaclass derived from this metaclass). As a result, class variables present in any given class will also be present in any class derived from this class (that is, class variables are inherited). In contrast, staticdata variables are introduced by a class (not a metaclass) and are (only) accessed from the class's **ClassData** structure — they are *not* inherited.

To declare an instance variable that is *not* a valid IDL type, a dummy typedef can be declared before the **interface** declaration and a **passthru** statement then used to pass the real typedef to the language-specific binding file(s). See also the section "Using the tk_foreign TypeCode" in Chapter 7, "The Interface Repository Framework." In the following example, the generic SOM type **somToken** is used in the IDL file for the user's type "myRealType". The **passthru** statement then causes an appropriate #include statement to be emitted into the C/C++ binding file, so that the file defining type "MyRealType" will be included when the binding file processes:

```
typedef somToken myRealType;

interface myClass : SOMObject {
. . .
  implementation {
     myRealType myInstVar;
     passthru C_h = "#include <myTypes.h>";
  };
};
```

Comments within a SOM IDL file

SOM IDL supports both C and C++ comment styles. The characters "//" start a line comment, which finishes at the end of the current line. The characters "/*" start a block comment that finishes with the "*/". Block comments do not nest. The two comment styles can be used interchangeably.

Comments in a SOM IDL specification must be strictly associated with particular syntactic elements, so that the SOM Compiler can put them at the appropriate place in the header and implementation files it generates. Therefore, comments may appear only in these locations (in general, following the syntactic unit being commented):

- At the beginning of the IDL specification
- After a semicolon
- Before or after the opening brace of a module, interface statement, implementation statement, structure definition, or union definition
- After a comma that separates parameter declarations or enumeration members
- After the last parameter in a prototype (before the closing parenthesis)
- After the last enumeration name in an enumeration definition (before the closing brace)
- After the colon following a case label of a union definition
- After the closing brace of an interface statement

Numerous examples of the use of comments can be found in the Tutorial of Chapter 2.

Because comments appearing in a SOM IDL specification are transferred to the files that the SOM Compiler generates, and because these files are often used as input to a programming language compiler, it is best within the body of comments to avoid using characters that are not generally allowed in comments of most programming languages. For example, the C language does not allow "*/" to occur within a comment, so its use is to be avoided, even when using C++ style comments in the .idl file.

SOM IDL also supports throw-away comments. They may appear anywhere in an IDL specification, because they are ignored by the SOM Compiler and are not transferred to any file it generates. Throw-away comments start with the string “//#” and end at the end of the line. Throw-away comments can be used to “comment out” portions of an IDL specification.

To disable comment processing (that is, to prevent the SOM Compiler from transferring comments from the IDL specification to the binding files it generates), use the **–c** option of the **sc** or **somc** command when running the SOM Compiler (See Section 4.3, “The SOM Compiler”). When comment processing is disabled, comment placement is not restricted, and comments can appear anywhere in the IDL specification.

Designating ‘private’ methods and attributes

To designate methods or attributes within an IDL specification as “private,” the declaration of the method or attribute must be surrounded with the preprocessor commands **#ifdef __PRIVATE__** (with two leading underscores and two following underscores) and **#endif**. For example, to declare a method “foo” as a private method, the following declaration would appear within the interface statement:

```
#ifdef __PRIVATE__
void foo();
#endif
```

Any number of methods and attributes can be designated as private, either within a single **#ifdef** or in separate ones. [Kernighan–Ritchie reference for the C preprocessor: pages 88-92.]

When compiling a .idl file, the SOM Compiler normally recognizes only public (nonprivate) methods and attributes, as that is generally all that is needed. To generate header files for client programs that do need to access private methods and attributes, the **–p** option should be included when running the SOM Compiler. The resulting .h or .xh header file will then include bindings for private, as well as public, methods and attributes. The **–p** option is described in the topic “Running the SOM Compiler” later in this chapter.

The SOMobjects Toolkit also provides a **pdl** (Public Definition Language) emitter that can be used with the SOM Compiler to generate a copy of a .idl file which has the portions designated as private removed. The next main section of this chapter describes how to invoke the SOM Compiler and the various emitters.

Defining multiple interfaces in a .idl file

A single .idl file can define *multiple interfaces*. This allows, for example, a class and its metaclass to be defined in the same file. When a file defines two (or more) interfaces that reference one another, forward declarations can be used to declare the name of an interface before it is defined. This is done as follows:

interface *className* **;**

The actual definition of the **interface** for “className” must appear later in the same .idl file.

If multiple interfaces are defined in the same .idl file, and the classes are not a class–metaclass pair, they can be grouped into modules, by using the following syntax:

module *moduleName* **{** *definition+* **};**

where each “definition” is a **type** declaration, **constant** declaration, **exception** declaration, **interface** statement, or nested **module** statement. Modules are used to scope identifiers (see below).

Alternatively, multiple interfaces can be defined in a single .idl file without using a module to group the interfaces. Whether or not a module is used for grouping multiple interfaces, the languages bindings produced from the .idl file will include support for all of the defined interfaces.

Note: When multiple interfaces are defined in a single .idl file and a **module** statement is not used for grouping these interfaces, it is necessary to use the **functionprefix** modifier to assure that different names exist for functions that provide different implementations for a method. In general, it is a good idea to always use the **functionprefix** modifier, but in this case it is essential.

Scoping and name resolution

A .idl file forms a *naming scope* (or *scope*). **Modules**, **interface** statements, **structures**, **unions**, **methods**, and **exceptions** form *nested scopes*. An identifier can only be defined once in a particular scope. Identifiers can be redefined in nested scopes.

Names can be used in an unqualified form within a scope, and the name will be resolved by successively searching the enclosing scopes. Once an unqualified name is defined in an enclosing scope, that name cannot be redefined.

Fully qualified names are of the form:

*scoped–name***::***identifier*

For example, method name "meth" defined within interface "Test" of module "M1" would have the fully qualified name:

```
M1::Test::meth
```

A qualified name is resolved by first resolving the "scoped-name" to a particular scope, S, and then locating the definition of "identifier" within that scope. Enclosing scopes of S are not searched.

Qualified names can also take the form:

::*identifier*

These names are resolved by locating the definition of "identifier" within the smallest enclosing module.

Every name defined in an IDL specification is given a global name, constructed as follows:

- Before the SOM Compiler scans a .idl file, the name of the current *root* and the name of the current *scope* are empty. As each module is encountered, the string "::" and the module name are appended to the name of the current root. At the end of the module, they are removed.
- As each interface, struct, union, or exception definition is encountered, the string "::" and the associated name are appended to the name of the current scope. At the end of the definition, they are removed. While parameters of a method declaration are processed, a new unnamed scope is entered so that parameter names can duplicate other identifiers.
- The global name of an IDL definition is then the concatenation of the current root, the current scope, a "::", and the local name for the definition.

The names of types, constants, and exceptions defined by the parents of a class are accessible in the child class. References to these names must be unambiguous. Ambiguities can be resolved by using a scoped name (prefacing the name with the name of the class that defines it and the characters "::", as in "parent-class::identifier"). Scope names can also be used to refer to a constant, type, or exception name defined by a parent class but redefined by the child class.

Name usage in client programs

Within a C or C++ program, the global name for a **type, constant,** or **exception** corresponding to an IDL scoped name is derived by converting the string "::" to an underscore ("_") and removing the leading underscore. Such names are referred to as *C–scoped names*. This means that types, constants, and exceptions defined within the interface statement for a class can be referenced in a C/C++ program by prepending the class name to the name of the type, constant, or exception. For example, consider the types defined in the following IDL specification:

```
typedef sequence<long,10> mySeq;
interface myClass : SOMObject
{
   enum color {red, white, blue};
   typedef string<100> longString;
   ...
}
```

These types could be accessed within a C or C++ program with the following global names:

```
mySeq,
myClass_color,
myClass_red,
myClass_white,
myClass_blue, and
myClass_longString
```

Type, constant, and exception names defined within modules similarly have the module name prepended. When using SOM's C/C++ bindings, the short form of type, constant, and exception names (such as, `color, longString`) can also be used where unambiguous, except that enumeration names must be referred to using the long form (for example: `myClass_red` and not simply `red`).

Because replacing "::" with an underscore to create global names can lead to ambiguity if an IDL identifier contains underscores, it is best to avoid the use of underscores when defining IDL identifiers.

Extensions to CORBA IDL permitted by SOM IDL

The following topics describe several SOM-unique extensions of the standard CORBA syntax that are permitted by SOM IDL for convenience. These constructs can be used in a .idl file without generating a SOM Compiler error.

If you want to verify that an IDL file contains only standard CORBA specifications, the SOM Compiler option **–mcorba** turns off each of these extensions and produces compiler errors wherever non-CORBA specifications are used. (The SOM Compiler command and options are described in the topic "Running the SOM Compiler" later in this chapter.)

Pointer '' types*

In addition to the base CORBA types, SOM IDL permits the use of pointer types ('*****'). As well as increasing the range of base types available to the SOM IDL programmer, using pointer types also permits the construction of more complex data types, including self-referential and mutually recursive structures and unions.

If self-referential structures and unions are required, then, instead of using the CORBA approach for IDL sequences, such as the following:

```
struct X {
     ...
     sequence <X> self;
     ...
};
```

it is possible to use the more typical C/C++ approach. For example:

```
struct X {
     ...
     X *self;
     ...
};
```

SOM IDL does not permit an explicit '*' in sequence declarations. If a sequence is required for a pointer type, then it is necessary to typedef the pointer type before use. For example:

```
sequence <long *> long_star_seq;        // error.

typedef long * long_star;
sequence <long_star> long_star_seq;      // OK.
```

Unsigned types

SOM IDL permits the syntax "unsigned <*type*>", where <*type*> is a previously declared type mapping onto "short" or "long". (Note that CORBA permits only "unsigned short" and "unsigned long".

Implementation section

SOM IDL permits an **implementation** section in an IDL **interface** specification to allow the addition of instance variables, method overrides, metaclass information, passthru information, and "pragma–like" information, called **modifiers**, for the emitters. See the topic "Implementation statements" earlier in this chapter.

Comment processing

The SOM IDL Compiler by default does not remove comments in the input source; instead, it attaches them to the nearest preceding IDL statement. This facility is useful, since it allows comments to be emitted in header files, C template files, documentation files, and so forth. However, if this capability is desired, this does mean that comments cannot be placed with quite as much freedom as with an ordinary IDL compiler. To turn off comment processing so that you can compile .idl files containing comments placed anywhere, you can use the compiler option **–c** or use "throw-away" comments throughout the .idl file (that is, comments preceded by //#); as a result, no comments will be included in the output files.

Generated header files

CORBA expects one header file, <*file*>.h, to be generated from <*file*>.idl. However, SOM IDL permits use of a class modifier, **filestem,** that changes this default output file name. (See "Running the SOM Compiler" later in this chapter.)

4.3 The SOM Compiler

The SOM Compiler translates the IDL definition of a SOM class into a set of "binding files" appropriate for the language that will implement the class's methods and the language(s) that will use the class. These bindings make it more convenient for programmers to implement and use SOM classes. The SOM Compiler currently produces binding files for the C and C++ languages.

Important Note: C and C++ bindings *cannot* both be generated during the same execution of the SOM compiler.

Generating binding files

The SOM Compiler operates in two phases:

- A precompile phase, in which a precompiler analyzes an OIDL or IDL class definition, and
- An emission phase, in which one or more emitter programs produce binding files.

Each binding file is generated by a separate emitter program. Setting the SMEMIT environment variable determines which emitters will be used, as described below. Note: In the discussion below, the <filestem> is determined by default from the name of the source .idl file with the ".idl"extension removed. Otherwise, a "filestem" modifier can be defined in the .idl file to specify another file name (see "Modifier statements" above).

Note: When changes to definitions in the .idl file later become necessary, the SOM Compiler should be rerun to update the current implementation template file, provided that the **c** or **xc** emitter is specified (either with the **–s** option or the SMEMIT environment variable, as described below). For more information on generating updates, see "Running incremental updates of the implementation template file" in Chapter 5, "Implementing Classes in SOM."

The emitters for the C language produce the following binding files:

*<filestem>***.c** — (produced by the **c** emitter)

This is a template for a C source program that implements a class's methods. This will become the primary source file for the class. (The other binding files can be generated from the **.idl** file as needed.) This template implementation file contains "stub" procedures for each method introduced or overridden by the class. (The stub procedures are empty of code except for required initialization and debugging statements.)

After the class implementor has supplied the code for the method procedures, running the **c** emitter again will update the implementation file to reflect changes made to the class definition (in the **.idl** file). These updates include adding new stub procedures, adding comments, and changing method prototypes to reflect changes made to the method definitions in the IDL specification. Existing code within method procedures is *not* disturbed, however.

The **.c** file contains an **#include** directive for the **.ih** file, described below.

The contents of the C source template is controlled by the Emitter Framework file <SOMBASE>/include/ctm.efw. This file can be customized to change the template produced. For detailed information on changing the template file see the *Emitter Framework Guide and Reference*.

<*filestem*>**.h** — (produced by the **h** emitter)

This is the header file to be included by C client programs (programs that use the class). It contains the C usage bindings for the class, including macros for accessing the class's methods and a macro for creating new instances of the class. This header file includes the header files for the class's parent classes and its metaclass, as well as the header file that defines SOM's generic C bindings, **som.h.**

<*filestem*>**.ih** — (produced by the **ih** emitter)

This is the header file to be included in the implementation file (the file that implements the class's methods — the .c file). It contains the implementation bindings for the class, including:

- a **struct** defining the class's instance variables,
- macros for accessing instance variables,
- macros for invoking parent methods the class overrides,
- the <*className*>**GetData** macro used by the method procedures in the <*filestem*>**.c** file (see "Stub procedures for methods" in Section 5.4 of Chapter 5),
- a <*className*>**NewClass** procedure for constructing the class object at run time, and
- any IDL types and constants defined in the IDL interface.

The emitters for the C++ language produce the following binding files:

<*filestem*>**.C** (for AIX) or <*filestem*>**.cpp** (for OS/2 or Windows) — (produced by the **xc** emitter)

This is a template for a C++ source program that implements a class's methods. This will become the primary source file for the class. (The other binding files can be generated from the **.idl** file as needed.) This template implementation file contains "stub" procedures for each method introduced or overridden by the class. (The stub procedures are empty of code except for required initialization and debugging statements.)

After the class implementor has supplied the code for the method procedures, running the **xc** emitter again will update this file to reflect changes made to the class definition (in the **.idl** file). These updates include adding new stub procedures, adding comments, and changing method prototypes to reflect changes made to the method definitions in the IDL specification. Existing code within method procedures is *not* disturbed, however.

The C++ implementation file contains an **#include** directive for the **.xih** file, described below.

The contents of the C++ source template is controlled by the Emitter Framework file <SOMBASE>/include/ctm.efw. This file can be customized to change the template produced. For detailed information on changing the template file see the *Emitter Framework Guide and Reference*.

*<filestem>***.xh** — (produced by the **xh** emitter)

This is the header file to be included by C++ client programs that use the class. It contains the usage bindings for the class, including a C++ definition of the class, macros for accessing the class's methods, and the **new** operator for creating new instances of the class. This header file includes the header files for the class's parent classes and its metaclass, as well as the header file that defines SOM's generic C++ bindings, **som.xh**.

*<filestem>***.xih** — (produced by the **xih** emitter)

This is the header file to be included in the implementation file (the file that implements the class's methods). It contains the implementation bindings for the class, including:

- a **struct** defining the class's instance variables,
- macros for accessing instance variables,
- macros for invoking parent methods the class overrides,
- the *<className>***GetData** macro (see "Stub procedures for methods" in Section 5.4 of Chapter 5),
- a *<className>***NewClass** procedure for constructing the class object at run time, and
- any IDL types and constants defined in the IDL interface.

Other files the SOM Compiler generates:

*<filestem>***.hh** — (produced by the **hh** emitter)

This file is a DirectToSOM C++ header file that describes a SOMobjects class in a way appropriate for DTS C++. When running this emitter, you must include the **noqualifytypes** command-line modifier for the **–m** option of the SOM Compiler command **sc** or **somc**.

*<filestem>***.pdl** — (produced by the **pdl** emitter)

This file is the same as the .idl file from which it is produced except that all items within the .idl file that are marked as "private" have been removed. (An item is marked as private by surrounding it with "#ifdef __PRIVATE__" and "#endif" directives.) Thus, the **pdl** (Public Definition Language) emitter can be used to generate a "public" version of the .idl file.

*<filestem>***.def** (for OS/2 or Windows)

— (produced by the **def** emitter)

This file is used by the linker to package a class as a library. To combine several classes into a single library, you must merge the **exports** statements from each of their **.def** files into a single **.def** file for the entire library. When packaging multiple classes in a single library, you must also write a simple C procedure named **SOMInitModule** and add it to the export list. This procedure should call the routine *<className>***NewClass** for each class packaged in the library. The **SOMInitModule** procedure is called by the SOM Class Manager when the library is dynamically loaded.

<*filestem*>**.exp** (for AIX)	—	(produced by the **exp** emitter) This file is used by the linker to package a class as a library. To combine several classes into a single library, you must merge the **exports** statements from each of their **.exp** files into a single **.exp** file for the entire library. When packaging multiple classes in a single library, you must also write a simple C procedure named **SOMInitModule** and add it to the export list. This procedure should call the routine <*className*>**NewClass** for each class packaged in the library. The **SOMInitModule** procedure is called by the SOM Class Manager when the library is dynamically loaded.
The **Interface Repository**		(produced by the **ir** emitter) See Chapter 7 for a discussion on the Interface Repository.

Note: The C/C++ bindings generated by the SOM Compiler have the following limitation: If two classes named “ClassName” and “ClassNameC” are defined, the bindings for these two classes will clash. That is, if a client program uses the C/C++ bindings (includes the .h/.xh header file) for both classes, a name conflict will occur. Thus, class implementors should keep this limitation in mind when naming their classes.

SOM users can extend the SOM Compiler to generate additional files by writing their own emitters. To assist users in extending the SOM Compiler, SOM provides an Emitter Framework — a collection of classes and methods useful for writing object-oriented emitters that the SOM Compiler can invoke. For more information, see the *Emitter Framework Guide and Reference*.

Note re: porting SOM classes: The header files (binding files) that the SOM Compiler generates will only work on the platform (operating system) on which they were generated. Thus, when porting SOM classes from the platform where they were developed to another platform, the header files must be regenerated from the .idl file by the SOM Compiler on that target platform.

Environment variables affecting the SOM Compiler

To execute the SOM Compiler on one or more files that contain IDL specifications for one or more classes, use the **sc** or **somc** command, as follows:

sc [–*options*] *files* (on AIX or OS/2)
somc [–*options*] *files* (on Windows)

where “files” specifies one or more .idl files.

Available “–options” for the command are detailed in the next topic. The operation of the SOM Compiler (whether it produces C binding files or C++ binding files, for example) is also controlled by certain environment variables that can be set before the **sc** or **somc** command is issued. The applicable environment variables are as follows:

SMEMIT — Determines which output files the SOM Compiler produces. Its value consists of a list of items separated by semicolons for OS/2, or by semicolons or colons for AIX. Each item designates an emitter to execute. For example, the statement:

```
SET SMEMIT=c;h;ih              (for OS/2)
export SMEMIT="c;h;ih"         (for AIX)
```

directs the SOM Compiler to produce the C binding files “hello.c”, “hello.h”, and“hello.ih” from the “hello.idl” input specification. By comparison,

```
SET SMEMIT=xc;xh;xih           (for OS/2)
export SMEMIT="xc;xh;xih"      (for AIX)
```

directs the SOM Compiler to produce C++ binding files "hello.C" (for AIX) or "hello.cpp" (for OS/2), "hello.xh", and "hello.xih" from the "hello.idl" input specification.

By default, all output files are placed in the same directory as the input file. If the SMEMIT environment variable is not set, then a default value of "h;ih" is assumed.

Windows note: The SMEMIT environmental variable can be set by using the SET command before the **somc** command is issued. For example:

```
SET SMEMIT="c;h;ih"
```
(for Windows)

If you are running the SOM Compiler from a DOS box under Windows, make sure to define SMEMIT *before* Windows is started.

SMINCLUDE — Specifies where the SOM Compiler should look for .idl files #included by the .idl file being compiled. Its value should be one or more directory names separated by a semicolon when using OS/2 or Windows, or separated by a semicolon or colon when using AIX. Directory names can be specified with absolute or relative pathnames. For example:

```
SET SMINCLUDE=.;..\MYSCDIR;C:\TOOLKT20\C\INCLUDE;
```
(for OS/2 or Windows)

```
export SMINCLUDE=.:myscdir:/u/som/include
```
(for AIX)

The default value of the SMINCLUDE environment variable is the "include" subdirectory of the directory into which SOM has been installed.

SMTMP — Specifies the directory that the SOM Compiler should use to hold intermediate output files. This directory should *not* coincide with the directory of the input or output files. For AIX, the default setting of SMTMP is /tmp; for OS/2 or Windows, the default setting of SMTMP is the root directory of the current drive.

OS/2 or Windows example:

```
SET SMTMP=..\MYSCDIR\GARBAGE
```

tells the SOM Compiler to place the temporary files in the GARBAGE directory.

Or, on OS/2 only:

```
SET SMTMP=%TMP%
```

tells the SOM Compiler to use the same directory for temporary files as given by the setting of the TMP environment variable (the default location for temporary system files). (On Windows, you cannot set one variable to another.)

AIX example:

```
export SMTMP=$TMP
export SMTMP=../myscdir/garbage
```

SMKNOWNEXTS — Specifies additional emitters to which the SOM Compiler should add a header. For example, if you were to write a new emitter for Pascal, called “emitpas”, then by default the SOM Compiler would not add any header comments to it. However, by setting SMKNOWNEXTS=pas, as shown:

```
set SMKNOWNEXTS=pas          (for OS/2 or Windows)
export SMKNOWNEXTS=pas       (for AIX)
```

the SOM Compiler will add a header to files generated with the “emitpas” emitter. The “header” added is a SOM Compiler-generated message plus any comments, such as copyright statements, that appear at the head of your .idl input file. For details on writing your own emitter, see the *Emitter Framework Guide and Reference*.

SOMIR — Specifies the name (or list of names) of the Interface Repository file. The **ir** emitter, if run, creates the Interface Repository, or checks it for consistency if it already exists. If the **–u** option is specified when invoking the SOM Compiler, the **ir** emitter also updates an existing Interface Repository.

SMADDSTAR — When defined, causes all interface references to have a “*****” added to them for the C bindings. The command-line options **–maddstar** and **–mnoaddstar** supercede and override the SMADDSTAR setting, however.

Running the SOM Compiler

The syntax of the command for running the SOM Compiler takes the forms:

sc [–*options*] *files* (on AIX or OS/2)
somc [–*options*] *files* (on Windows)

The “files” specified in the **sc** or **somc** command denote one or more files containing the IDL class definitions to be compiled. If no extension is specified, .idl is assumed. By default, the <filestem> of the .idl file determines the filestem of each emitted file. Otherwise, a “filestem” modifier can be defined in the .idl file to specify another name (see “Modifier statements” discussed earlier).

Selected “–options” can be specified individually, as a string of option characters, or as a combination of both. Any option that takes an argument either must be specified individually or must appear as the final option in a string of option characters. Available options and their purposes are as follows:

–C *n* Sets the maximum allowable size for a simple comment in the .idl file (default: 32767). This is only needed for very large single comments.

–D *name*[=*def*]
Defines *name* as in a #define directive. The default *def* is 1. This option is the same as the –D option for the C compiler. Note: This option can be used to define **__PRIVATE__** so that the SOM Compiler will also compile any methods and attributes that have been defined as private using the directive **#ifdef __PRIVATE__**; however, the –p option does the same thing more easily.

–E *variable=value*

Sets an environment variable. (See the previous topic for a discussion of the available environment variables: SMEMIT, SMINCLUDE, SMTMP, and SMNOIR.)

–I *dir* When looking for #included files, looks first in dir, then in the standard directories (same as the C compiler **–I** option).

–S *n* Sets the total allowable amount of unique string space used in the IDL specification for names and passthru lines (default: 32767). This is only needed for very large .idl files.

–U *name* Removes any initial definition (via a #define preprocessor directive) of symbol *name*.

–V Displays version information about the SOM Compiler.

–c Turns off comment processing. This allows comments to appear anywhere within an IDL specification (rather than in restricted places), and it causes comments not to be transferred to the output files that the SOM Compiler produces.

–d *directory* Specifies a directory where all output files should be placed. If the –d option is not used, all output files are placed in the same directory as the input file.

–h or **–?** Produces a listing of this option list. (This option is typically used in an **sc** or **somc** command that does not include a .idl file name).

–i *filename* Specifies the name of the class definition file. Use this option to override the built-in assumption that the input file will have a .idl extension. Any *filename* supplied with the –i option is used exactly as it is specified.

–m *name*[**=***value*]

Adds a global modifier. The currently supported global modifiers are as follows:

addprefixes

Adds 'functionprefixes' to the method procedure prototypes during an incremental update of the implementation template file. This option applies only when rerunning the **c** or **xc** emitter on an IDL file that previously did *not* specify a functionprefix. A class implementor who later decides to use prefixes should add a line in the 'implementation' section of the .idl file containing the specification:

functionprefix = *prefix*

(as described earlier in the topic "Modifier statements") and then rerun the c or xc emitter using the **–maddprefixes** option. The method procedure prototypes in the implementation file will then be updated so that each method name includes the assigned prefix. (This option does not support changes to existing prefix names, nor does it apply for OIDL files.)

addstar This option causes all interface references to have a '*****' added to them for the C bindings. See the earlier section entitled "Object types" for further details.

comment=*comment string*

where *comment string* can be either of the designations: "/*" or "//". This option indicates that comments marked in the designated manner in the .idl file are to be completely ignored by the SOM Compiler and will *not* be included in the output files. <u>Note</u>: Comments on lines beginning with "//#" are always ignored by the SOM Compiler.

–m *name*[**=***value*] (modifier options continued from the previous page)

corba This option directs the SOM Compiler to compile the input definition according to strict CORBA-defined IDL syntax. This means, for example, that comments may appear anywhere and that pointers are not allowed. When the **–mcorba** option is used, parts of a .idl file surrounded by **#ifdef __SOMIDL__** and **#endif** directives are ignored. This option can be used to determine whether all nonstandard constructs (those specific to SOM IDL) are properly protected by **#ifdef __SOMIDL__** and **#endif** directives.

csc This option forces the OIDL compiler to be run. This is required only if you want to compile an OIDL file that does not have an extension of .csc or .sc.

emitappend
This option causes emitted files to be appended at the end of existing files of the same name.

noaddstar
This option ensures that interface references will not have a "*" added to them for the C bindings. This is the default setting; it is the opposite of the **–m** compiler option **addstar**.

noheader
This option ensures that the SOM Compiler does not add a header to the beginning of an emitted file.

noint This option directs the SOM Compiler not to warn about the portability problems of using `int`'s in the source.

nolock This option causes the Interface Repository Emitter **emitir** (see Chapter 7, "Interface Repository Framework") to leave the IR unlocked when updates are made to it. This can improve performance on networked file systems. By not locking the IR, however, there is the risk of multiple processes attempting to write to the same IR, with unpredictable results. This option should only be used when you know that only one process is updating an IR at once.

nopp This option directs the SOM Compiler not to run the SOM preprocessor on the .idl input file.

noqualifytypes
This option prevents the use of C-scoped names in emitter output, and is used in conjunction with the .hh emitter.

notc This option directs the SOM Compiler not to create TypeCode information when emitting files. This is required only when the .idl files contain some undeclared types. This option is typically used when compiling converted .csc files that have not had typing information added.

nouseshort
This option directs the SOM Compiler not to generate short forms for type names in the .h and .xh public header files. This can be useful to save disk space.

pp=*preprocessor*
This option directs the SOM Compiler to use the specified preprocessor as the SOM preprocessor, rather than the default "somcpp". Any standard C/C++ preprocessor can be used as a preprocessor for IDL specifications.

tcconsts This option directs the SOM Compiler to generate TypeCode constants in the .h and .xh public header files. Please refer to the Interface Repository (described in Chapter 7) for more details.

Note: All command-line **–m** modifier options can be specified in the environment by changing them to UPPERCASE and prepending "SM" to them. For example, if you want to always set the options "–mnotc" and "–maddstar", set corresponding environment variables as follows:

On OS/2 or Windows:

```
set SMNOTC=1
set SMADDSTAR=1
```

On AIX:

```
export SMNOTC=1
export SMADDSTAR=1
```

–p Causes the "private" sections of the IDL file to be included in the compilation (that is, sections preceded by #ifdef __PRIVATE__ that contain private methods and attributes). Note: The –p option is equivalent to the earlier option –D__PRIVATE__.

–r Checks that all names specified in the release order statement are valid method names (default: FALSE).

–s *string* Substitutes *string* in place of the contents of the **SMEMIT** environment variable for the duration of the current **sc** or **somc** command. This determines which emitters will be run and, hence, which output files will be produced. (If a list of values is given, on OS/2 *only* the list must be enclosed in double quotes.)

The –s option is a convenient way to override the SMEMIT environment variable. In OS/2 for example, the command:

```
> SC –s"h;c" EXAMPLE
```

is equivalent to the following sequence of commands:

```
> SET OLDSMEMIT=%SMEMIT%
> SET SMEMIT=H;C
> SC EXAMPLE
> SET SMEMIT=%OLDSMEMIT%
```

Similarly, in AIX the command:

```
> sc –sh";"c example
```

is equivalent to the following sequence of commands:

```
> export OLDSMEMIT=$SMEMIT
> export SMEMIT=h";"c
> sc example
> export SMEMIT=$OLDSMEMIT
```

Windows users: You should *not* use quotes in substitution strings of the **–s** option for the **somc** command:

`> somc –sh;ih *.idl`	Correct
`> somc –s"h;ih" *.idl`	Incorrect

–u Updates the Interface Repository (default: no update). With this option, the Interface Repository will be updated even if the **ir** emitter is not explicitly requested in the SMEMIT environment variable or the **–s** option.

–v Uses verbose mode to display informational messages (default: FALSE). This option is primarily intended for debugging purposes and for writers of emitters.

–w Suppresses warning messages (default: FALSE).

The following sample commands illustrate various options for the **sc** command (or similarly with **somc**):

`sc –sc hello.idl`	Generates file "hello.c".
`sc –hV`	Generates a help message and displays the version of the SOM Compiler currently available.
`sc –vsh";"ih hello.idl`	Generates "hello.h" and "hello.ih" with informational messages. Note: On Windows, omit the quotes `(")` when issuing this command.
`sc –sxc –doutdir hello.idl`	Generates "hello.xc" in directory "outdir".

4.4 The 'pdl' Facility

As discussed earlier in this chapter, the SOM Compiler provides a **pdl** (Public Definition Language) emitter. This emitter generates a file that is the same as the .idl file from which it is produced, except that it removes all items within the .idl file that are marked as "private." An item is marked as private by surrounding it with "#ifdef __PRIVATE__" and "#endif" directives. Thus, the **pdl** emitter can be used to generate a "public" version of a .idl file. (Generally, client programs will need only the "public" methods and attributes.)

The SOMobjects Toolkit also provides a separate program, **pdl**, which performs the same function as the **pdl** emitter, but can be invoked independently of the SOM Compiler. In addition, the **pdl** program can remove any kind of items in the .idl file that are preceded by a user-specified "#ifdef" directive and followed by an "#endif" directive. The **pdl** program is invoked as follows:

pdl [–c | d | f | h | s | /] *files*

where "files" specifies one or more .idl files whose specified "#ifdef" sections are to be removed. Filenames must be completely specified (with the .idl extension). If no "#ifdef" directive is specified (by including a **–**/<string> option), then the "#ifdef __PRIVATE__" sections will be removed by default.

The **pdl** command supports the following options. (Selected options can be specified individually, as a string of option characters, or as a combination of both. Any option that takes an argument either must be specified individually or must appear as the final option in a string of option characters.)

–c *cmd*	Specifies that, for each .idl file, the **pdl** program is to run the specified system command. This command may contain a single occurrence of the string "%s", which will be replaced with the source file name before the command is executed. For example the option –c"sc –sh %s" has the same effect as issuing the **sc** command with the –sh option.
–d *dir*	Specifies a directory in which the output files are to be placed. (The output files are given the same name as the input files.) If no directory is specified, the output files are named <*fileStem*>.pdl (where *fileStem* is the file stem of the input file) and are placed in the current working directory.
–h	Requests this description of the **pdl** command syntax and options.
–f	Specifies that output files are to replace existing files with the same name, even if the existing files are read–only. By default, files are replaced only if they have write access.
–s *smemit*	Specifies the SMEMIT variable with which the **pdl** program is to invoke the SOM Compiler.
–/ <string>	Specifies the "#ifdef" pattern for which the **pdl** program will strip out .idl statements. The default is "#ifdef __PRIVATE__".

For example, to install public versions of the .idl files in the directory "pubinclude", type:

```
pdl –d pubinclude *.idl
```

Chapter 5. Implementing Classes in SOM

Contents

Chapter 5. Implementing Classes in SOM

This chapter begins with a more in-depth discussion of SOM concepts and the SOM run-time environment than was appropriate in Chapter 2, “Tutorial for Implementing SOM Classes.” Subsequent sections then provide information about completing an implementation template file, updating the template file, compiling and linking, packaging classes in libraries, and other useful topics for class implementors. During this process, you can refer to Chapter 4, “SOM IDL and the SOM Compiler,” if you want to read the reference information or see the full syntax related to topics discussed in this chapter. The current chapter ends with topics describing how to customize SOMobjects execution in various ways.

5.1 The SOM Run-time Environment

As discussed in Chapter 1, the SOMobjects Developer Toolkit provides

- The **SOM Compiler**, used when creating SOM class libraries, and
- The **SOM run-time library**, for using SOM classes at execution time.

The SOM run-time library provides a set of *functions* used primarily for creating objects and invoking methods on them. The *data structures* and *objects* that are created, maintained, and used by the functions in the SOM run-time library constitute the **SOM run-time environment**.

A distinguishing characteristic of the SOM run-time environment is that SOM *classes* are represented by run-time *objects*; these objects are called *class objects*. By contrast, other object-oriented languages such as C++ treat classes strictly as compile-time structures that have no properties at run time. In SOM, however, each class has a corresponding run-time object. This has three advantages: First, application programs can access information about a class at run time, including its relationships with other classes, the methods it supports, the size of its instances, and so on. Second, because much of the information about a class is established at run time rather than at compile time, application programs needn’t be recompiled when this information changes. Finally, because class objects can be instances of user-defined classes in SOM, users can adapt the techniques for subclassing and inheritance in order to build object-oriented solutions to problems that are otherwise not easily addressed within an OOP context.

Run-time environment initialization

When the SOM run-time environment is initialized, four primitive SOM objects are automatically created. Three of these are class objects (**SOMObject**, **SOMClass**, and **SOMClassMgr**), and one is an instance of **SOMClassMgr**, called the **SOMClassMgrObject**. Once loaded, application programs can invoke methods on these class objects to perform tasks such as creating other objects, printing the contents of an object, freeing objects, and the like. These four primitive objects are discussed below.

In addition to creating the four primitive SOM objects, initialization of the SOM run-time environment also involves initializing global variables to hold data structures that maintain the state of the environment. Other functions in the SOM run-time library rely on these global variables.

For application programs written in C or C++ that use the language-specific bindings provided by SOM, the SOM run-time environment is automatically initialized the first time any object is created. Programmers using other languages must initialize the run-time environment explicitly by calling the **somEnvironmentNew** function (provided by the SOM run-time library) before using any other SOM functions or methods.

SOMObject class object

SOMObject is the root class for all SOM *classes*. It defines the essential *behavior* common to all SOM *objects*. All user-defined SOM classes are derived, directly or indirectly, from this class. That is, every SOM class is a subclass of **SOMObject** or of some other class derived from **SOMObject**. **SOMObject** has no instance variables, thus objects that inherit from **SOMObject** incur no size increase. They do inherit a suite of methods that provide the behavior required of all SOM objects.

SOMClass class object

Because SOM classes are run-time objects, and since all run-time objects are instances of some class, it follows that a SOM class object must also be an instance of some class. The class of a class is called a *metaclass*. Hence, the instances of an ordinary class are individuals (nonclasses), while the instances of a metaclass are class objects.

In the same way that the class of an object defines the "instance methods" that the object can perform, the metaclass of a class defines the "class methods" that the class itself can perform. *Class methods* (sometimes called *factory methods* or *constructors*) are performed by class objects. Class methods perform tasks such as creating new instances of a class, maintaining a count of the number of instances of the class, and other operations of a "supervisory" nature. Also, class methods facilitate inheritance of instance methods from parent classes.

The distinction between instance methods and class methods, as well as that between objects, classes, and metaclasses, is illustrated in Figure 1. For information on the distinction between parent classes and metaclasses, see the section "Parent class vs. metaclass," below.

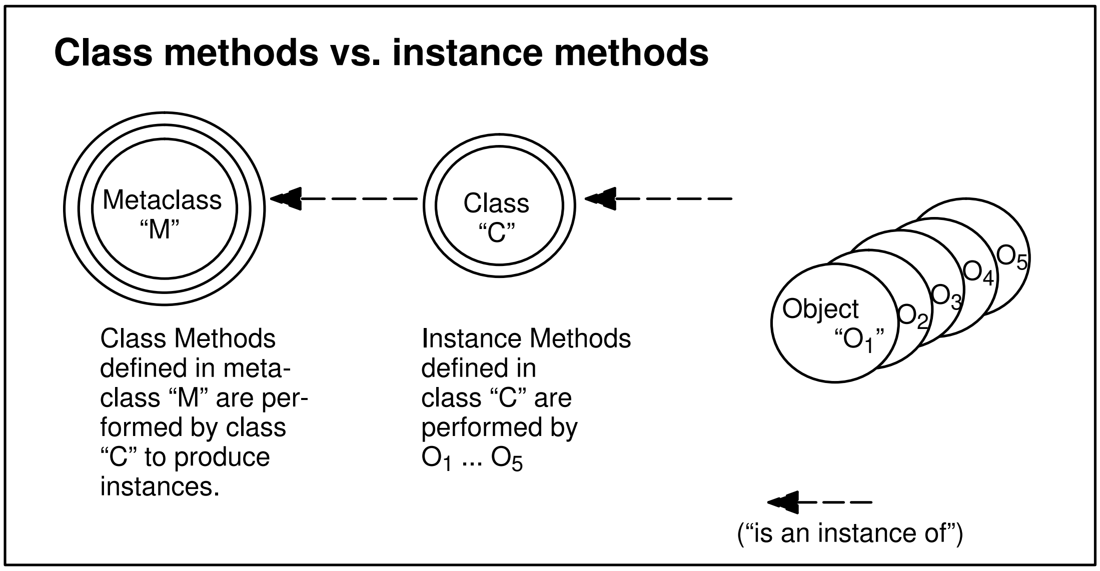


Figure 1. Class methods vs. instance methods.

SOMClass is the root class for all SOM *metaclasses*. That is, all SOM metaclasses must be subclasses of **SOMClass** or of some metaclass derived from **SOMClass**. **SOMClass** defines the essential behavior common to all SOM *class objects*. In particular, **SOMClass** provides:

- Six class methods for creating new class instances: **somNew**, **somNewNoInit**, **somRenew**, **somRenewNoInit**, **somRenewNoZero**, and **somRenewNoInitNoZero**.
- A number of class methods that dynamically obtain or update information about a class and its methods at run time, including:
  – **somInitMIClass,** for implementing multiple inheritance from parent classes,
  – **somOverrideSMethod,** for overriding inherited methods, and
  – **somAddStaticMethod** and **somAddDynamicMethod**, for introducing new methods.

SOMClass is a subclass (or child) of **SOMObject**. Hence, SOM *class objects* can also perform the same set of basic *instance methods* common to all SOM objects. This is what allows SOM classes to be real objects in the SOM run-time environment. **SOMClass** also has the unique distinction of being its own metaclass (that is, **SOMClass** defines its own class methods).

A user-defined class can designate as its metaclass either **SOMClass** or another user-written metaclass descended from **SOMClass**. If a metaclass is not explicitly specified, SOM determines one automatically.

SOMClassMgr class object and SOMClassMgrObject

The third primitive SOM class is **SOMClassMgr**. A single instance of the **SOMClassMgr** class is created automatically during SOM initialization. This instance is referred to as the **SOMClassMgrObject**, because it is pointed to by the *global variable* SOMClassMgrObject**.** The object **SOMClassMgrObject** has the responsibility to

- Maintain a registry (a run-time directory) of all SOM classes that exist within the current process, and to
- Assist in the dynamic loading and unloading of class libraries.

For C/C++ application programs using the SOM C/C++ language bindings, the **SOMClassMgrObject** automatically loads the appropriate library file and constructs a run-time object for the class the first time an instance of a class is created. For programmers using other languages, **SOMClassMgr** provides a method, **somFindClass**, for directing the **SOMClassMgrObject** to load the library file for a class and create its class object.

Relationships among the four primitive SOM run-time objects are illustrated in Figure 2. Again, the primitive classes supplied with SOM are **SOMObject, SOMClass,** and **SOMClassMgr.** During SOM initialization, the latter class generates an instance called **SOMClassMgrObject.** The left-hand side of Figure 2 shows parent-class relationships between the built-in SOM classes, and the right-hand side shows instance/class relationships. That is, on the left **SOMObject** is the parent class of **SOMClass** and **SOMClassMgr**. On the right **SOMClass** is the metaclass of itself, of **SOMObject,** and of **SOMClassMgr**, which are all class objects at run time. **SOMClassMgr** is the class of **SOMClassMgrObject**.

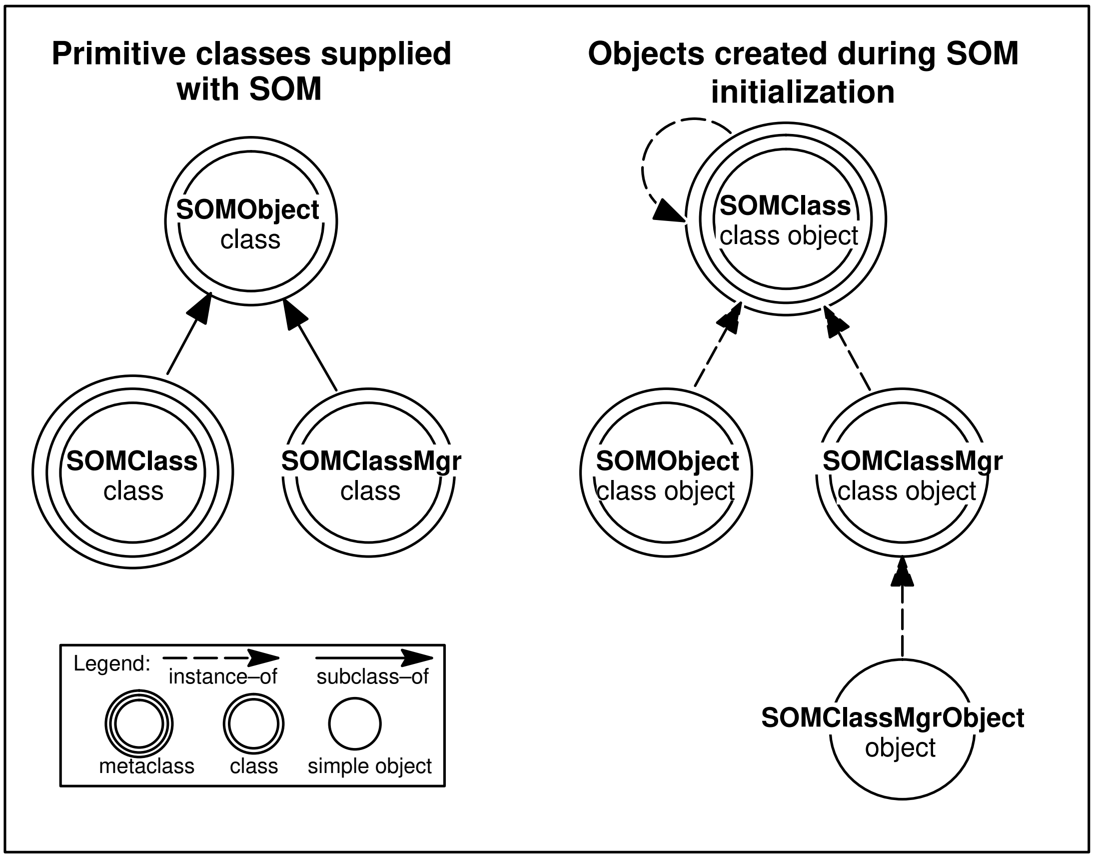


Figure 2. The SOM run-time environment provides four primitive objects, three of them class objects.

Parent class vs. metaclass

There is a distinct difference between the notions of "parent" (or base) class and "metaclass." Both notions are related to the fact that a class defines the methods and variables of its instances, which are therefore called *instance methods* and *instance variables*.

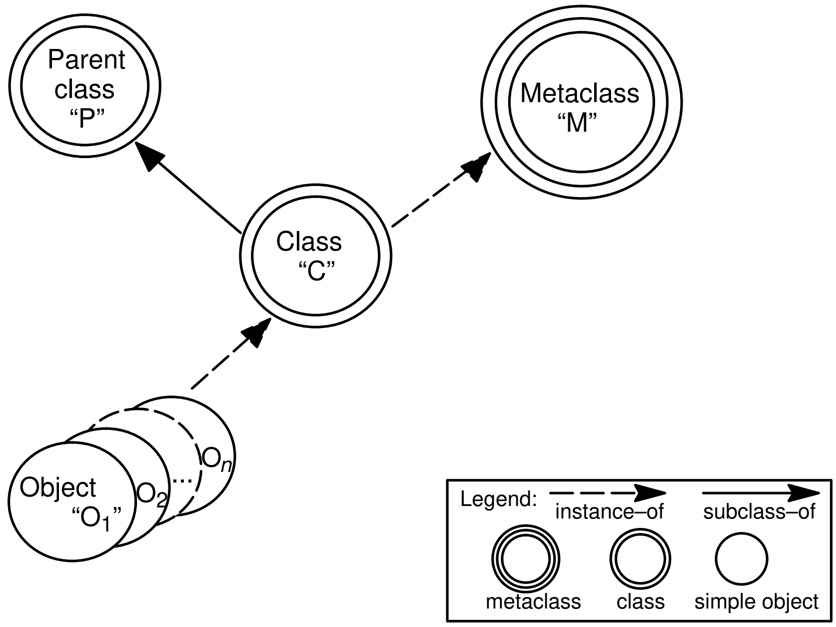
A *parent* of a given class is a class from which the given class is *derived* by *subclassing*. (Thus, the given class is called a *child* or a *subclass* of the parent.) A parent class is a class from which instance methods and instance variables are *inherited*. For example, the parent of class "Dog" might be class "Animal". Hence, the instance methods and variables introduced by "Animal" (such as methods for breathing and eating, or a variable for storing an animal's weight) would also apply to instances of "Dog", because "Dog" *inherits* these from "Animal", its parent class. As a result, any given dog instance would be able to breath and eat, and would have a weight.

A *metaclass* is a class whose instances are class objects, and whose instance methods and instance variables (as described above) are therefore the methods and variables of class objects. For this reason, a *metaclass* is said to define *class methods* — the methods that a class object performs. For example, the metaclass of "Animal" might be "AnimalMClass", which defines the methods that can be invoked on class "Animal" (such as, to create Animal instances — objects that are not classes, like an individual pig or cat or elephant or dog).

Note: It is important to distinguish the methods of a class object (that is, the methods that can be invoked on the class object, which are defined by its metaclass) from the methods that the class defines for its instances.

To summarize: the *parent* of a class provides inherited methods that the class's *instances* can perform; the *metaclass* of a class provides class methods that the *class itself* can perform. The distinctions between parent class and metaclass are summarized in Figure 3.

Characteristics of parent class vs. metaclass



Any class "C" has both a *metaclass* and one or more *parent* class(es).

- The *parent* class(es) of "C" provide the inherited *instance methods* that individual instances (objects "O_i") of class "C" can perform. Instance methods that an instance "O_i" performs might include (a) initializing itself, (b) performing computations using its instance variables, (c) printing its instance variables, or (d) returning its size. New instance methods are defined by "C" itself, in addition to those inherited from C's parent classes.
- The *metaclass* "M" defines the *class methods* that class "C" can perform. For example, class methods defined by metaclass "M" include those that allow "C" to (a) inherit its parents's instance methods and instance variables, (b) tell its own name, (c) create new instances, and (d) tell how many instance methods it supports. These methods are inherited from SOMClass. Additional methods supported by "M" might allow "C" to count how many instances it creates.
- Each class "C" has one or more parent classes and exactly one metaclass. (The single exception is SOMObject, which has no parent class.) Parent class(es) must be explicitly identified in the IDL declaration of a class. (SOMObject is given as a parent if no subsequently-derived class applies.) If a metaclass is not explicitly listed, the SOM run time will determine an applicable metaclass.
- An instance of a metaclass is always another *class object*. For example, class "C" is an instance of metaclass "M". SOMClass is the SOM-provided metaclass from which all subsequent metaclasses are derived.

Figure 3. A class has both parent classes and a metaclass.

A metaclass has its own inheritance hierarchy (through its parent classes) that is independent of its instances' inheritance hierarchies. In Figure 4, a sequence of classes is defined (or derived), stemming from **SOMObject**. The child class (or subclass) at the end of this line ("C_2") inherits instance methods from all of its ancestor classes (here, **SOMObject** and "C_1"). An instance created by "C2" can perform any of these instance methods. In an analogous manner, a line of metaclasses is defined, stemming from **SOMClass.** Just as a new class is derived from an existing class (such as **SOMObject**), a new metaclass is derived from an existing metaclass (such as **SOMClass**). In this example, both **SOMObject** and class "C1" are instances of the **SOMClass** metaclass, whereas class "C2" is an instance of metaclass "M2", which inherits from **SOMClass**.

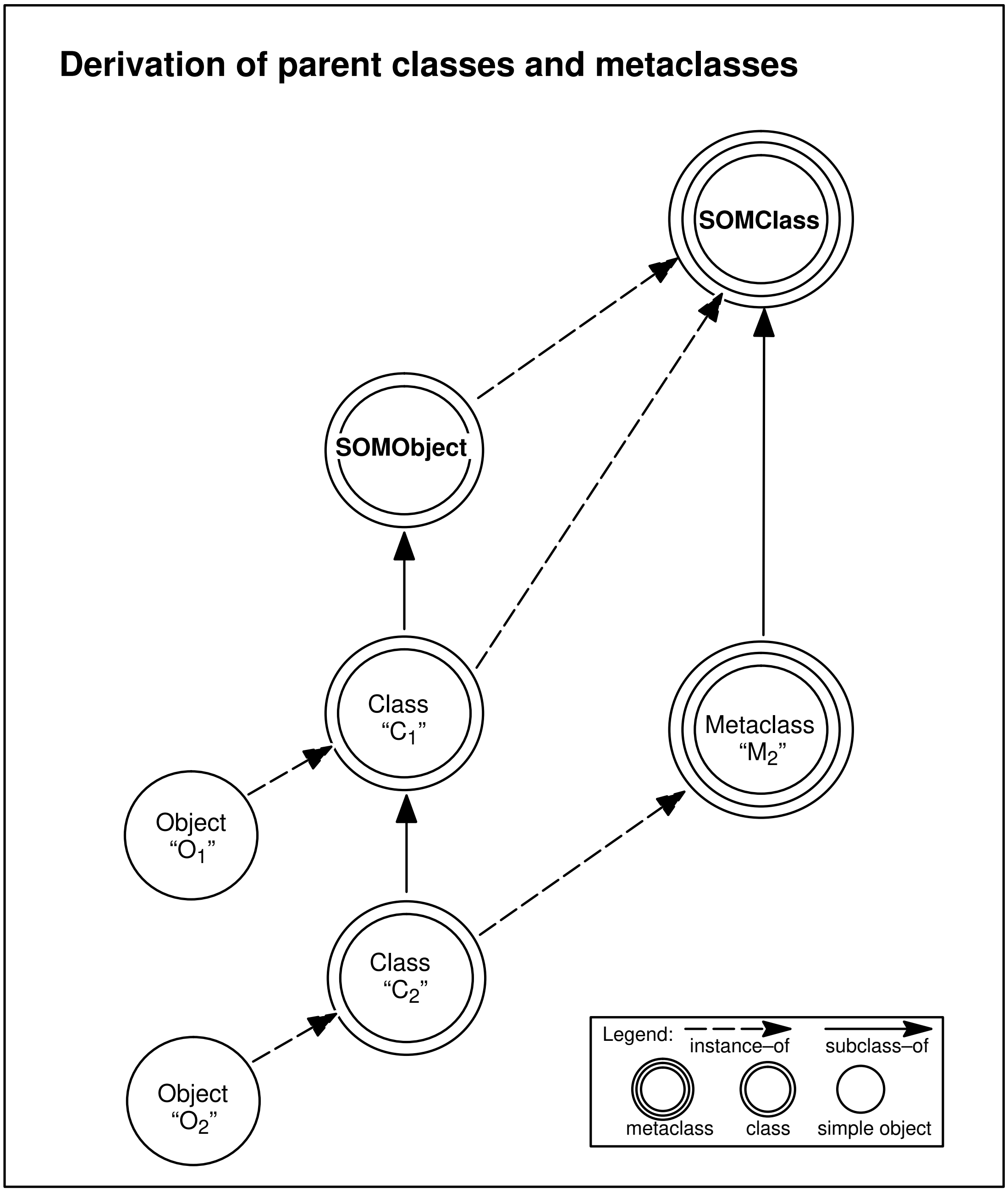


Figure 4. Parent classes and metaclasses each have their own independent inheritance hierarchies.

SOM-derived metaclasses

As previously discussed, a class object can perform any of the class methods that its metaclass defines. New metaclasses are typically created to modify existing class methods or introduce new class method(s). Chapter 10, “Metaclass Framework,” discusses metaclass programming.

Three factors are essential for effective use of metaclasses in SOM:

- First, every class in SOM is an object that is implemented by a metaclass.
- Second, programmers can define and name new metaclasses, and can use these metaclasses when defining new SOM classes.
- Finally, and most importantly, metaclasses cannot interfere with the fundamental guarantee required of every OOP system: specifically, any code that executes without method-resolution error on instances of a given class will also execute without method-resolution errors on instances of any subclass of this class.

Surprisingly, SOM is currently the only OOP system that can make this final guarantee while also allowing programmers to explicitly define and use named metaclasses. This is possible because SOM automatically determines an appropriate metaclass that supports this guarantee, automatically deriving new metaclasses by subclassing at run time when this is necessary.

To better understand this concept, consider the situation in Figure 5. Here, class “A” is an instance of metaclass “AMeta”. Assume that “AMeta” supports a method “bar” and that “A” supports a method “foo” that uses the expression “_bar(_somGetClass(somSelf)).” That is, method “foo” invokes “bar” on the class of the object on which “foo” is invoked. For example, when method “foo” is invoked on an instance of class “A” (say, object “O_1”), this in turn invokes “bar” on class “A” itself.

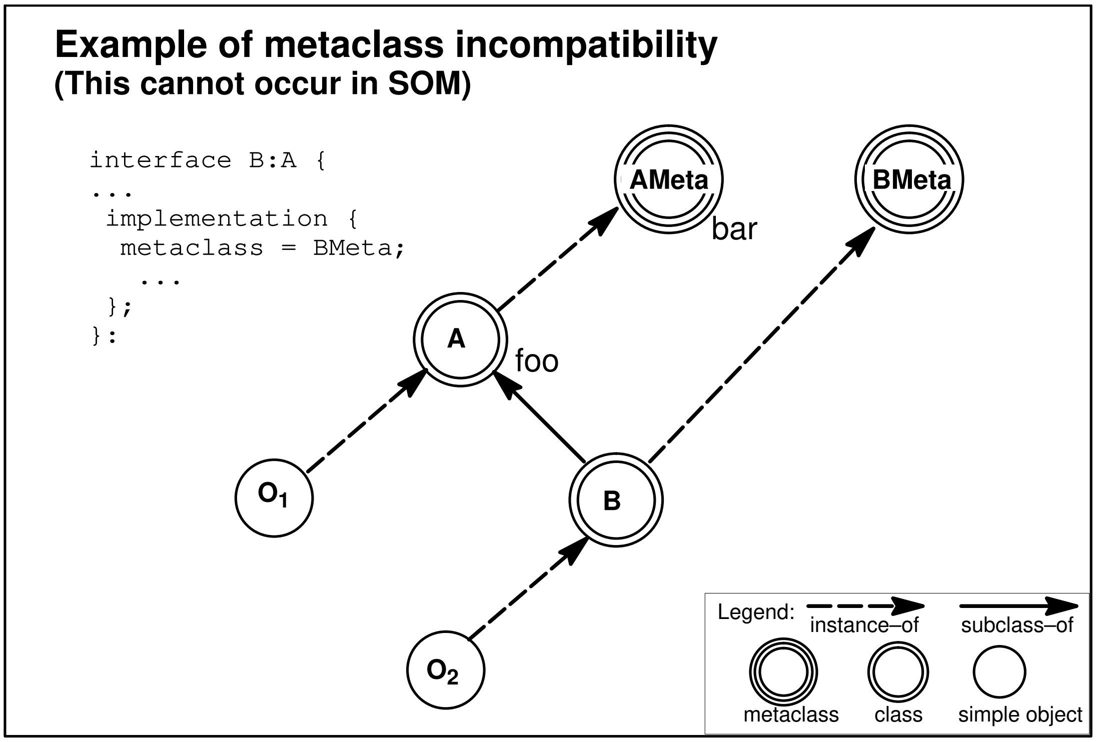


Figure 5. Example of Metaclass Incompatibility.

Now consider what happens when “A” is subclassed by “B”, a class that has the explicit metaclass “BMeta” declared in its SOM IDL source file, as shown by the code in Figure 5. If the class hierarchy were formed as in Figure 5, then an invocation of “foo” on “O_2” would fail, because metaclass “BMeta” does not support the “bar” method introduced by “AMeta”.

There is only one way that “BMeta” can support this specific method — by inheriting it from “AMeta” (“BMeta” could introduce another method named “bar”, but this would be a *different* method from the one introduced by “AMeta”). Therefore, in this example, because “BMeta” is not a subclass of “AMeta”, “BMeta” cannot be allowed to be the metaclass of “B”. That is, “BMeta” is not compatible with the requirements placed on “B” by the fundamental principle of OOP referred to above. This situation is referred to as *metaclass incompatibility*.

SOM does not allow hierarchies with metaclass incompatibilities. Instead, SOM automatically builds *derived metaclasses* when this is necessary. The resulting class hierarchy in this example is depicted in Figure 6, where SOM has automatically built the metaclass “DerivedMetaclass”. This ensures that the invocation of method “foo” on instances of class “B” will not fail, and also ensures that the desired class methods provided by “BMeta” will be available on class “B”.

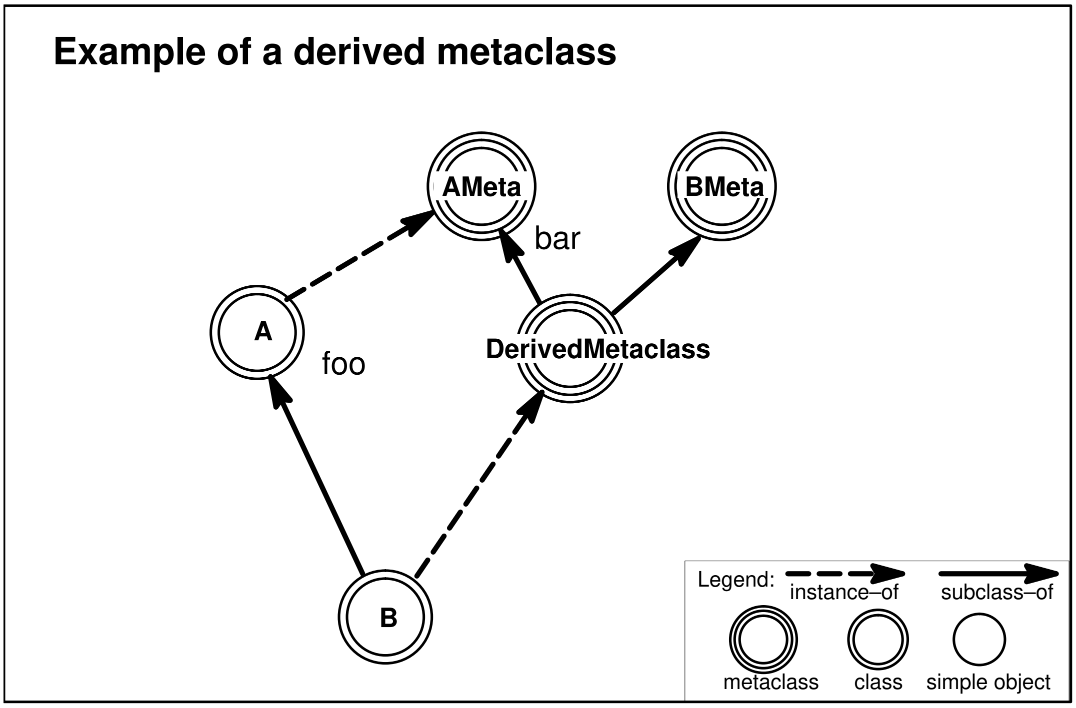


Figure 6. Example of a Derived Metaclass.

There are three important aspects of SOM’s approach to derived metaclasses:

- First, the creation of SOM-derived metaclasses is integrated with programmer-specified metaclasses. If a programmer-specified metaclass already supports all the class methods and variables needed by a new class, then the programmer-specified metaclass will be used as is.
- Second, if SOM must derive a different metaclass than the one explicitly indicated by the programmer (in order to support all the necessary class methods and variables), then the SOM-derived metaclass inherits from the explicitly indicated metaclass first. As a result, the method procedures defined by the specified metaclass take precedence over other possibilities (see the following section on inheritance and the discussion of resolution of ambiguity in the case of multiple inheritance).
- Finally, the class methods defined by the derived metaclass invoke the appropriate initialization methods of its parents to ensure that the class variables of its instances are correctly initialized.

As further explanation for the automatic derivation of metaclasses, consider the following multiple-inheritance example. In Figure 7, class “C” does not have an explicit metaclass declaration in its SOM IDL, yet its parents do. As a result, class “C” requires a derived metaclass. (If you still have trouble following the reasoning behind derived metaclasses, ask yourself the following question: What class should “C” be an instance of? After a bit of reflection, you will conclude that, if SOM did not build the derived metaclass, you would have to do so yourself.)

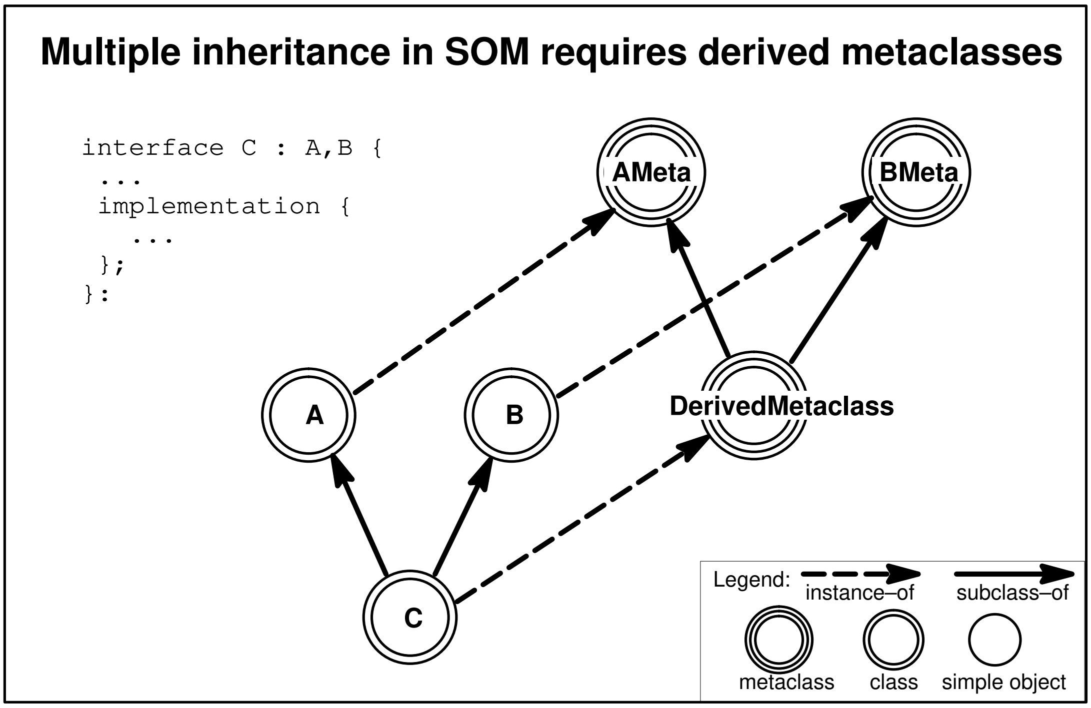


Figure 7. Multiple Inheritance requires Derived Metaclasses.

In summary, SOM allows and encourages the definition and explicit use of named metaclasses. With named metaclasses, programmers can not only affect the behavior of class instances by choosing the parents of classes, but they can also affect the behavior of the classes themselves by choosing their metaclasses. Because the behavior of classes in SOM includes the implementation of inheritance itself, metaclasses in SOM provide an extremely flexible and powerful capability allowing classes to package solutions to problems that are otherwise very difficult to address within an OOP context.

At the same time, SOM is unique in that it relieves programmers of the responsibility for avoiding metaclass incompatibility when defining a new class. At first glance, this might seem to be merely a useful (though very important) convenience. But, in fact, it is absolutely essential, because SOM is predicated on binary compatibility with respect to changes in class implementations.

A programmer might, at one point in time, know the metaclasses of all ancestor classes of a new subclass, and, as a result, be able to explicitly derive an appropriate metaclass for the new class. Nevertheless, SOM must guarantee that this new class will still execute and perform correctly when any of its ancestor class’s implementations are changed (which could even include specifying different metaclasses). Derived metaclasses allow SOM to make this guarantee. A SOM programmer need never worry about the problem of metaclass incompatibility; SOM does this for the programmer. Instead, explicit metaclasses can simply be used to “add in” whatever behavior is desired for a new class. SOM automatically handles anything else that is needed. Chapter 10 provides useful examples of such metaclasses. A SOM programmer should find numerous applications for the techniques that are illustrated there.

5.2 Inheritance

One of the defining aspects of an object model is its characterization of inheritance. This section describes SOM's model for inheritance.

A class in SOM defines an implementation for objects that support a specific interface:

- The *interface* defines the methods supported by objects of the class, and is specified using SOM IDL.
- The *implementation* defines what instance variables implement an object's state and what procedures implement its methods.

New classes are derived (by subclassing) from previously existing classes through inheritance, specialization, and addition. Subclasses inherit interface from their parent classes: any method available on instances of a class is also available on instances of any class derived from it (either directly or indirectly). Subclasses also inherit implementation (the procedures that implement the methods) from their parent classes *unless* the methods are *overridden* (redefined or specialized). In addition, a subclass may introduce new instance methods and instance variables that will be inherited by other classes derived from it.

SOM also supports *multiple inheritance*. That is, a class may be derived from (and may inherit interface and implementation from) multiple parent classes. Note: Multiple inheritance is available only to SOM classes whose interfaces are specified in IDL, and not to SOM classes whose interfaces are specified in SOM's earlier interface definition language, OIDL. See Appendix B for information on how to automatically convert existing OIDL files to IDL.

It is possible under multiple inheritance to encounter potential conflicts or ambiguities with respect to inheritance. All multiple inheritance models must face these issues, and resolve the ambiguities in some way. For example, when multiple inheritance is allowed, it is possible that a class will inherit the same method or instance variable from different parents (because each of these parents has some common ancestor that introduces the method or instance variable). In this situation, a SOM subclass inherits only one implementation of the method or instance variable. (The implementation of an instance variable within an object is just the location where it is stored. The implementation of a method is a procedure pointer, stored within a method table.) The following illustration addresses the question of which method implementation would be inherited.

Consider the situation in Figure 8. Class "W" defines a method "foo", implemented by procedure "proc1". Class "W" has two subclasses, "X" and "Y". Subclass "Y" overrides the implementation of "foo" with procedure "proc2". Subclass "X" does not override "foo". In addition, classes "X" and "Y" share a common subclass, "Z". That is, the IDL interface statement for class "Z" lists its parents as "X" and "Y" in that order.

The question is thus: which implementation of method "foo" does class "Z" inherit — procedure "proc1" defined by class "W", or procedure "proc2" defined by class "Y"? The procedure for performing inheritance that is defined by SOMClass resolves this ambiguity by using the *left path precedence* rule: when the same method is inherited from multiple ancestors, the procedure used to support the method is the one used by the *leftmost ancestor* from which the method is inherited. (The ordering of parent classes is determined by the order in which the class implementor lists the parents in the IDL specification for the class.)

In Figure 8, then, class "Z" inherits the implementation of method "foo" defined by class "W" (procedure "proc1"), rather than the implementation defined by class "Y" (procedure "proc2"), because "X" is the leftmost ancestor of "Z" from which the method "foo" is inherited. This rule may be interpreted as giving priority to classes whose instance interfaces are mentioned first in IDL interface definitions.

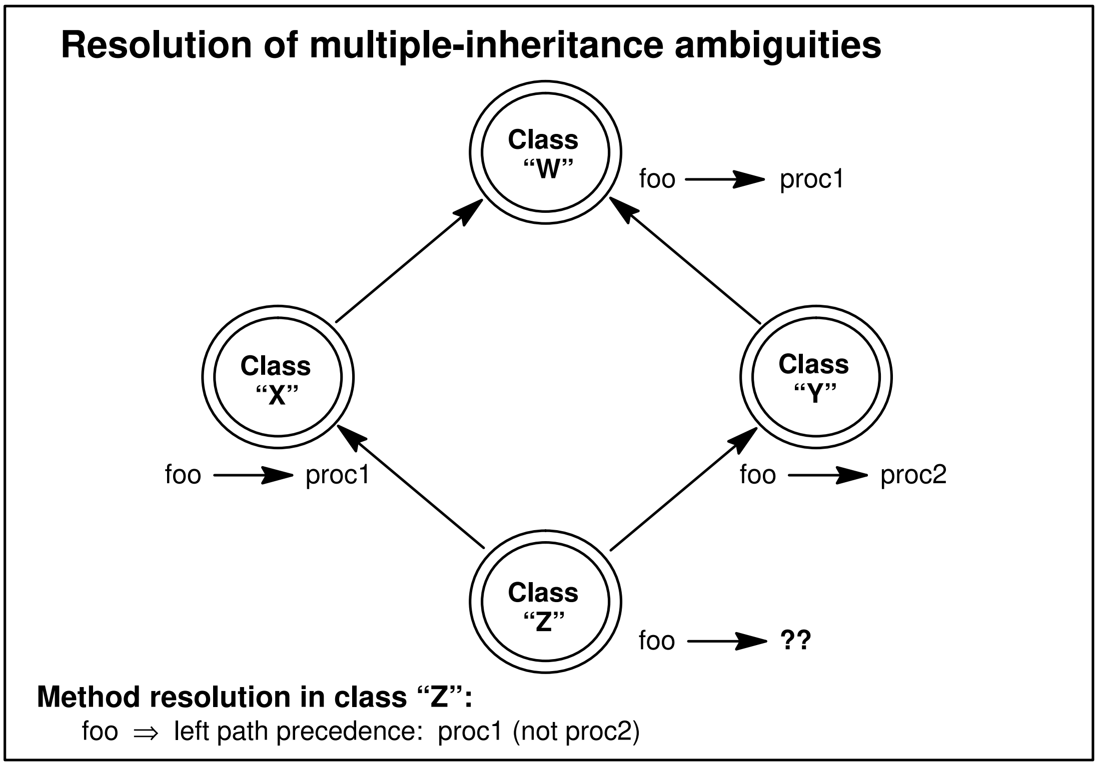


Figure 8. SOMClass uses the *left path precedence* rule to resolve some multiple inheritance ambiguities.

If a class implementor decides that the default inherited implementation is not appropriate (for example, procedure "proc2" is desired), then SOM IDL allows the class designer to select the parent whose implementation is desired. For more information concerning this approach, see the **select** modifier, which is documented in the topic "Modifier statements" in Chapter 4, "SOM IDL and the SOM Compiler."

Note: Alternatively, an explicit metaclass for class "Z" could be introduced to change the way methods are inherited. However, this would be a fairly serious step to take — it would also affect the semantics of inheritance for all of Z's descendant classes. Also, this would be done by overriding **somInitMIClass**, which is strongly discouraged until such time that SOMobjects includes the Cooperative Metaclass among its officially supported interface to the Metaclass Framework.

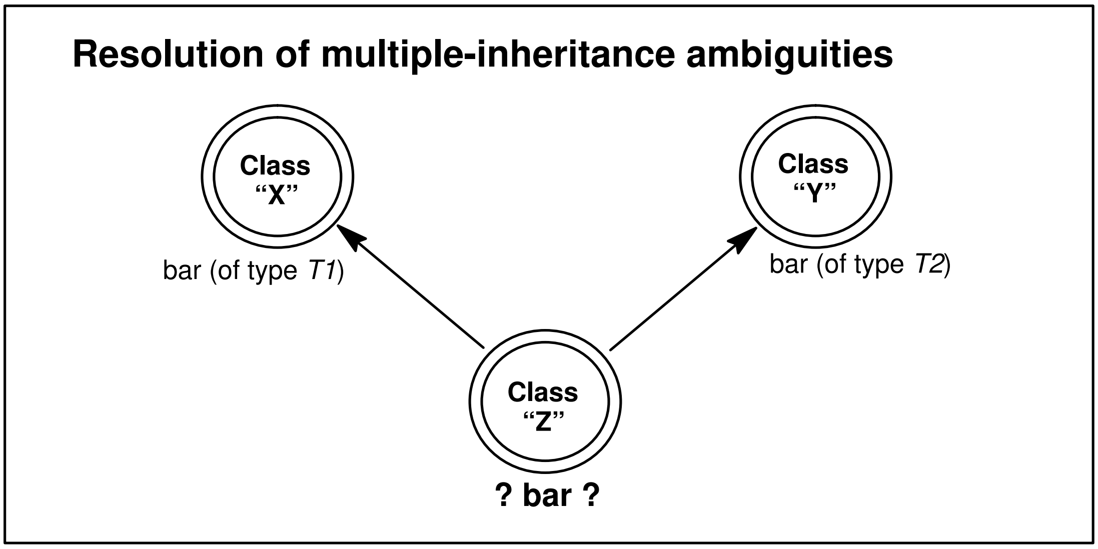


Figure 9. Some multiple inheritance ambiguities are illegal in IDL.

Another conflict that may arise with the use of multiple inheritance is when two ancestors of a class define different methods (in general, with different signatures) with the same name. For example, consider Figure 9. Class "X" defines a method "bar" with type *T1*, and class "Y" defines a method "bar" with type *T2*. Class "Z" is derived from both "X" and "Y", and "Z" does not override method "bar".

This example illustrates a method name that is "overloaded" — that is, used to name two entirely different methods (note that overloading is completely unrelated to overriding). This is not necessarily a difficult problem to handle. Indeed, the run-time SOM API allows the construction of a class that supports the two different "bar" methods illustrated in FIgure 9. (They are implemented using two different method-table entries, each of which is associated with its introducing class.)

<u>However, the interface to instances of such classes can*not* be defined using IDL</u>. IDL specifically forbids the definition of interfaces in which method names are overloaded. Furthermore, within SOM itself, the use of such classes can lead to anomalous behavior unless care is taken to avoid the use of name-lookup method resolution (discussed in the following section), since, in this case, a method name alone does not identify a unique method. For this reason, (statically declared) multiple-inheritance classes in SOM are currently restricted to those whose instance interfaces can be defined using IDL. Thus, the above example cannot be constructed with the aid of the SOM Compiler.

5.3 Method Resolution

Method resolution is the step of determining which procedure to execute in response to a method invocation. For example, consider this scenario:

- Class “Dog” introduces a method “bark”, and
- A subclass of “Dog”, called “BigDog”, overrides “bark”, and
- A client program creates an instance of either “Dog” or “BigDog” (depending on some run-time criteria) and invokes method “bark” on that instance.

Method resolution is the process of determining, at run time, which method procedure to execute in response to the method invocation (either the method procedure for “bark” defined by “Dog”, or the method procedure for “bark” defined by “BigDog”). This determination depends on whether the receiver of the method (the object on which it is invoked) is an instance of “Dog” or “BigDog” (or perhaps depending on some other criteria).

SOM allows class implementors and client programs considerable flexibility in deciding how SOM performs method resolution. In particular, SOM supports three mechanisms for method resolution, described in order of increased flexibility and increased computational cost: offset resolution, name-lookup resolution, and dispatch-function resolution.

Offset resolution

When using SOM’s C and C++ language bindings, offset resolution is the default way of resolving methods, because it is the fastest. For those familiar with C++, this is roughly equivalent to the C++ “virtual function” concept.

Although offset resolution is the fastest technique for method resolution, it is also the most constrained. Specifically, using offset resolution requires these constraints:

- The name of the method to be invoked must be known at compile time,
- The name of the class that introduces the method must be known at compile time (although not necessarily by the programmer), and
- The method to be invoked must be part of the introducing class’s static (IDL) interface definition.

To perform offset method resolution, SOM first obtains a *method token* from a global data structure associated with the class that introduced the method. This data structure is called the *ClassData structure*. It includes a method token for each method the class introduces. The method token is then used as an “index” into the receiver’s *method table,* to access the appropriate method procedure. Because it is known at compile time which class introduces the method and where in that class’s ClassData structure the method’s token is stored, offset resolution is quite efficient. The cost of offset method resolution is currently about twice the cost of calling a C function using a pointer loaded with the function address.

An object’s method table is a table of pointers to the procedures that implement the methods that the object supports. This table is constructed by the object’s class and is shared among the class instances. The method table built by class (for its instances) is referred to as the class’s *instance method table*. This is useful terminology, since, in SOM, a class is itself an object with a method table (created by its metaclass) used to support method calls on the class.

Usually, offset method resolution is sufficient; however, in some cases, the more flexible name-lookup resolution is required.

Name-lookup resolution

Name-lookup resolution is similar to the method resolution techniques employed by Objective-C and Smalltalk. It is currently about five times slower than offset resolution. It is more flexible, however. In particular, name-lookup resolution, unlike offset resolution, can be used when:

- The name of the method to be invoked isn't known until run time, or
- The method is added to the class interface at run time, or
- The name of the class introducing the method isn't known until run time.

For example, a client program may use two classes that define two different methods of the same name, and it might not be known until run time which of the two methods should be invoked (because, for example, it will not be known until run time which class's instance the method will be applied to).

Name-lookup resolution is performed by a class, so it requires a method call. (Offset resolution, by contrast, requires no method calls.) To perform name-lookup method resolution, the class of the intended receiver object obtains a method procedure pointer for the desired method that is appropriate for its instances. In general, this will require a name-based search through various data structures maintained by ancestor classes.

Offset and name-lookup resolution achieve the same net effect (that is, they select the same method procedure); they just achieve it differently (via different mechanisms for locating the method's method token). Offset resolution is faster, because it does not require searching for the method token, but name-lookup resolution is more flexible.

When defining (in SOM IDL) the interface to a class of objects, the class implementor can decide, for each method, whether the SOM Compiler will generate usage bindings that support name-lookup resolution for invoking the method. Regardless of whether this is done, however, application programs using the class can have SOM use either technique, on a per-method-call basis. Chapter 3, "Using SOM Classes in Client Programs," describes how client programs invoke methods."

Dispatch-function resolution

Dispatch-function resolution is the slowest, but most flexible, of the three method-resolution techniques. Dispatch functions permit method resolution to be <u>based on arbitrary rules associated with the class</u> of which the receiving object is an instance. Thus, a class implementor has complete freedom in determining how methods invoked on its instances are resolved.

With both offset and name-lookup resolution, the net effect is the same — the method procedure that is ultimately selected is the one supported by the class of which the receiver is an instance. For example, if the receiver is an instance of class "Dog", then Dog's method procedure will be selected; but if the receiver is an instance of class "BigDog", then BigDog's method procedure will be selected.

By contrast, dispatch-function resolution allows a class of instances to be defined such that the method procedure is selected using some other criteria. For example, the method procedure could be selected on the basis of the arguments to the method call, rather than on the receiver. For more information on dispatch-function resolution, see the description and examples for the **somDispatch**, and **somOverrideMtab** methods in the *SOMobjects Developer Toolkit Programmers Reference Manual*.

Customizing method resolution

Customizing method resolution requires the use of metaclasses that override SOMClass methods. This is not recommend without use of the Cooperative Metaclass that guarantees correct operation of SOMobjects in conjunction with such metaclasses. SOMobjects users who require this functionality should request access to the experimental Cooperative Metaclass used to implement the SOMobjects Metaclass Framework. Metaclasses implemented using the Cooperative Metaclass may have to be reprogrammed in the future when SOMobjects introduces an officially supported Cooperative Metaclass.

The four kinds of SOM methods

SOM supports four different kinds of methods: *static* methods, *nonstatic* methods, *dynamic* methods, and *direct-call procedures*. The following paragraphs explain these four method categories and the kinds of method resolution available for each.

Static methods

These are similar in concept to C++ virtual functions. Static methods are normally invoked using offset resolution via a method table, as described above, but all three kinds of method resolution are applicable to static methods. Each different static method available on an object is given a different slot in the object's method table. When SOMobjects Toolkit language bindings are used to implement a class, the SOM IDL **method** modifier can be specified to indicate that a given method is static; however, this modifier is rarely used since it is the default for SOM methods.

Static methods introduced by a class can be overridden (redefined) by any descendant classes of the class. When SOMobjects language bindings are used to implement a class, the SOM IDL **override** modifier is specified to indicate that a class overrides a given inherited method. When a static method is resolved using offset resolution, it is not important which interface is accessing the method — the actual class of the object determines the method procedure that is selected.

Note: All SOM IDL modifiers are described in the topic "Modifier statements" in Chapter 4, "SOM IDL and the SOM Compiler."

Nonstatic methods

These methods are similar in concept to C++ nonstatic member functions (that is, C++ functions that are not virtual member functions and are not static member functions). Nonstatic methods are normally invoked using offset resolution, but all three kinds of method resolution are applicable to nonstatic methods. When the SOMobjects language bindings are used to implement a class, the SOM IDL **nonstatic** modifier is used to indicate that a given method is nonstatic.

Like static methods, nonstatic methods are given individual positions in method tables. However, nonstatic methods cannot be overridden. Instead, descendants of a class that introduces a nonstatic method can use the SOM IDL **reintroduce** modifier to "hide" the original nonstatic method with another (nonstatic or static) method of the same name and signature. When a nonstatic method is resolved, selection of the specific method procedure is determined by the interface that is used to access the method.

Dynamic methods

These methods are *not* declared when specifying an object interface using IDL. Instead, they are registered with a class object at run time using the method **somAddDynamicMethod**. Because there is no way for SOM to know about dynamic methods before run time, offset resolution is not available for dynamic methods. Only name-lookup or dispatch-function resolution can be used to invoke dynamic methods. Dynamic methods cannot be overridden.

Direct-call procedures

These are similar in concept to C++ static member functions. Direct-call procedures are not given positions in SOM method tables, but are accessed directly from a class's **ClassData** structure. Strictly speaking, none of the previous method-resolution approaches apply for invoking a direct-call procedure, although SOMobjects language bindings provide the same invocation syntax for direct-call procedures as for static or nonstatic methods. Direct-call procedures cannot be overridden, but they can be reintroduced. When SOMobjects language bindings are used to implement a class, the SOM IDL **procedure** modifier is used to indicate that a given method is a direct-call procedure. Note: Methods having the **procedure** modifier cannot be invoked remotely using DSOM.

5.4 Implementing SOM Classes

The *interface* to a class of objects contains the information that a client must know to use an object — namely, the signatures of its methods and the names of its attributes. The interface is described in a formal language independent of the programming language used to implement the object's methods. In SOM, the formal language used to define object interfaces is the Interface Definition Language (described in Chapter 4, "SOM IDL and the SOM Compiler").

The *implementation* of a class of objects (that is, the procedures that implement the methods and the instance variables that store an object's state) is written in the implementor's preferred programming language. This language can be object-oriented (for instance, C++) or procedural (for instance, C).

A completely implemented class definition, then, consists of two main files:

- An IDL specification of the interface to instances of the class — the *interface definition file* (or .idl file) and
- Method procedures written in the implementor's language of choice — the *implementation file*.

The SOM Compiler provides the link between those two files: To assist users in implementing classes, the SOM Compiler produces a *template* implementation file — a type-correct guide for how the implementation of a class should look. Then, the class implementor modifies this template file to fully implement the class's methods. That process is the subject of the remainder of this chapter.

The SOM Compiler can also update the implementation file to reflect changes subsequently made to a class's interface definition file (the .idl file). These *incremental updates* include adding new stub procedures, adding comments, and changing method prototypes to reflect changes made to the method declarations in the IDL specification. These updates to the implementation file, however, do *not* disturb existing code in the method procedures. These updates are discussed further in "Running incremental updates of the implementation template file" later in this section.

For C programmers, the SOM Compiler generates a <*filestem*>**.c** file. For C++ programmers, the SOM Compiler generates a <*filestem*>**.C** file (for AIX) or a <*filestem*>**.cpp** file (for OS/2 or Windows). To specify whether the SOM Compiler should generate a C or C++ implementation template, set the value of the SMEMIT environment variable, or use the **–s** option when running the SOM Compiler. (See "The SOM Compiler" in Chapter 4, "SOM IDL and the SOM Compiler.")

Note: As this chapter describes, a SOM class can be implemented by using C++ to define the instance variables introduced by the class and to define the procedures that implement the overridden and introduced methods of the class. Be aware, however, that the C++ class defined by the C++ usage bindings for a SOM class (described in Chapter 3) *cannot* be subclassed in C++ to create new C++ or SOM classes.†

† The reason why the C++ implementation of a SOM class involves the definition of C++ procedures (not C++ methods) to support SOM methods is that there is no language-neutral way to call a C++ method. Only C++ code can call C++ methods, and this calling code must be generated by the same compiler that generates the method code. In contrast, the method procedures that implement SOM methods must be callable from any language, without knowledge on the part of the object client as to which language is used to implement the resolved method procedure.

The implementation template

Consider the following IDL description of the “Hello” class:

```
#include <somobj.idl>

interface Hello : SOMObject
{
    void sayHello();
    // This method outputs the string ”Hello, World!”.
};
```

From this IDL description, the SOM Compiler generates the following C implementation template, hello.c (a C++ implementation template, hello.C or hello.cpp, is identical except that the *#included* file is <hello.xih> rather than <hello.ih>):

```
#define Hello_Class_Source
#include <hello.ih>

/*
 *  This method outputs the string ”Hello, World!”.
 */

SOM_Scope void   SOMLINK sayHello(Hello somSelf, Environment *ev)
{
    /* HelloData *somThis = HelloGetData(somSelf); */
    HelloMethodDebug(”Hello”,”sayHello”);
}
```

The first line defines the “Hello_Class_Source” symbol, which is used in the SOM-generated implementation header files for C to determine when to define various functions, such as “HelloNewClass.” For interfaces defined within a module, the directive “#define <className>_Class_Source” is replaced by the directive “#define SOM_Module_<moduleName>_Source”.

The second line (`#include <hello.ih>` for C, or `#include <hello.xih>` for C++) includes the SOM-generated implementation header file. This file defines a **struct** holding the class’s instance variables, macros for accessing instance variables, macros for invoking parent methods, and so forth.

Stub procedures for methods

For each method introduced or overridden by the class, the implementation template includes a *stub procedure* — a procedure that is empty except for an *initialization* statement, a *debugging* statement, and possibly a *return* statement. The stub procedure for a method is preceded by any comments that follow the method’s declaration in the IDL specification.

For method “sayHello” above, the SOM Compiler generates the following prototype of the stub procedure:

```
SOM_Scope void SOMLINK sayHello(Hello somSelf, Environment *ev)
```

The “SOM_Scope” symbol is defined in the implementation header file as either “extern” or “static,” as appropriate. The term “void” signifies the return type of method “sayHello”. The “SOMLINK” symbol is defined by SOM; it represents the keyword needed to link to the C or C++ compiler, and its value is system-specific. Using the “SOMLINK” symbol allows the code to work with a variety of compilers without modification.

Following the “SOMLINK” symbol is the name of the procedure that implements the method. If no **functionprefix** modifier has been specified for the class, then the procedure name is the same as the method name. If a **functionprefix** modifier is in effect, then the procedure name is generated by prepending the specified prefix to the method name. For example, if the class definition contained the following statement:

```
functionprefix = xx_;
```

then the prototype of the stub procedure for method “sayHello” would be:

```
SOM_Scope void SOMLINK xx_sayHello(Hello somSelf, Environment *ev)
```

The **functionprefix** *cannot* be

```
<classname>_
```

since this is used in method invocation macros defined by the C usage bindings.

Following the procedure name is the formal parameter list for the method procedure. Because each SOM method always receives at least one argument (a pointer to the SOM object that responds to the method), the first parameter name in the prototype of each stub procedure is called **somSelf**. (The macros defined in the implementation header file rely on this convention.) The **somSelf** parameter is a pointer to an object that is an instance of the class being implemented (here, class “Hello”) or an instance of a class derived from it.

Unless the IDL specification of the class included the **callstyle=oidl** modifier, then the formal parameter list will include one or two additional parameters before the parameters declared in the IDL specification: an **(Environment *ev)** input/output parameter, which permits the return of exception information, and, if the IDL specification of the method includes a context specification, a **(Context *ctx)** input parameter. These parameters are prescribed by the CORBA standard. For more information on using the **Environment** and **Context** parameters, see the section entitled “Exceptions and error handling” in Chapter 3, “Using SOM Classes in Client Programs,” and the book *The Common Object Request Broker: Architecture and Specification*, published by Object Management Group and X/Open.

The first statement in the stub procedure for method “sayHello” is the statement:

```
/* HelloData *somThis = HelloGetData(somSelf); */
```

This statement is enclosed in comments only when the class does *not* introduce any instance variables. The purpose of this statement, for classes that do introduce instance variables, is to initialize a local variable (**somThis**) that points to a *structure* representing the instance variables introduced by the class. The **somThis** pointer is used by the macros defined in the “Hello” implementation header file to access those instance variables. (These macros are described below.) In this example, the “Hello” class introduces no instance variables, so the statement is commented out. If instance variables are later added to a class that initially had none, then the comment characters can be removed if access to the variable is required.

The “HelloData” type and the “HelloGetData” macro used to initialize the **somThis** pointer are defined in the implementation header file. Within a method procedure, class implementors can use the **somThis** pointer to access instance data, or they can use the convenience macros defined for accessing each instance variable, as described below.

To implement a method so that it can modify a local copy of an object’s instance data without affecting the object’s real instance data, declare a variable of type <*className*>**Data** (for example, “HelloData”) and assign to it the structure that **somThis** points to; then make the **somThis** pointer point to the copy. For example:

```
HelloData myCopy = *somThis;
somThis = &myCopy;
```

Next in the stub procedure for method “sayHello” is the statement:

```
HelloMethodDebug("Hello","sayHello");
```

This statement facilitates debugging. The “HelloMethodDebug” macro is defined in the implementation header file. It takes two arguments, a class name and a method name. If debugging is turned on (that is, if global variable **SOM_TraceLevel** is set to one in the calling program), the macro produces a message each time the method procedure is entered. (See Chapter 3, “Using SOM Classes in Client Programs,” for information on debugging with SOM.)

Debugging can be permanently disabled (regardless of the **SOM_TraceLevel** setting in the calling program) by redefining the <*className*>MethodDebug macro as **SOM_NoTrace(c,m)** following the #include directive for the implementation header file. (This can yield a slight performance improvement.) For example, to permanently disable debugging for the “Hello” class, insert the following lines in the hello.c implementation file following the line “#include hello.ih” (or “#include hello.xih,” for classes implemented in C++):

```
#undef HelloMethodDebug
#define HelloMethodDebug(c,m) SOM_NoTrace(c,m)
```

The way in which the stub procedure ends is determined by whether the method is a new or an overriding method.

- For non-overriding (new) methods, the stub procedure ends with a return statement (unless the return type of the method is **void**). The class implementor should customize this return statement.
- For overriding methods, the stub procedure ends by making a “parent method call” for each of the class’s parent classes. If the method has a return type that is not **void**, the last of these parent method calls is returned as the result of the method procedure. The class implementor can customize this return statement if needed (for example, if some other value is to be returned, or if the parent method calls should be made before the method procedure’s own processing). See the next section for a discussion of parent method calls.

If a **classinit** modifier was specified to designate a user-defined procedure that will initialize the “Hello” class object, as in the statement:

```
classinit = HInit;
```

then the implementation template file would include the following stub procedure for “HInit”, in addition to the stub procedures for Hello’s methods:

```
void  SOMLINK HInit(SOMClass *cls)
{

}
```

This stub procedure is then filled in by the class implementor. If the class definition specifies a **functionprefix** modifier, the **classinit** procedure name is generated by prepending the specified prefix to the specified **classinit** name, as with other stub procedures.

Extending the implementation template

To implement a method, add code to the body of the stub procedure. In addition to standard C or C++ code, class implementors can also use any of the functions, methods, and macros SOM provides for manipulating classes and objects. Chapter 3, “Using SOM Classes in Client Programs,” discusses these functions, methods, and macros.

In addition to the functions, methods, and macros SOM provides for both class clients and class implementors, SOM provides two facilities especially for class implementors. They are for (1) accessing instance variables of the object responding to the method and (2) making parent method calls, as follows.

Accessing internal instance variables

To access internal instance variables, class implementors can use either of the following forms:

_variableName

somThis–>*variableName*

To access internal instance variables "a", "b", and "c", for example, the class implementor could use either _a, _b, and _c, or somThis–>a, somThis–>b, and somThis–>c. These expressions can appear on either side of an assignment statement. The **somThis** pointer must be properly initialized in advance using the <*className*>**GetData** procedure, as shown above.

Instance variables can be accessed only within the implementation file of the class that introduces the instance variable, and not within the implementation of subclasses or within client programs. (To allow access to instance data from a subclass or from client programs, use an *attribute* rather than an instance variable to represent the instance data.) For C++ programmers, the *_variableName* form is available only if the macro **VARIABLE_MACROS** is defined (that is, **#define VARIABLE_MACROS**) in the implementation file prior to including the .xih file for the class.

Making parent method calls

In addition to macros for accessing instance variables, the implementation header file that the SOM Compiler generates also contains definitions of macros for making "parent method calls." When a class overrides a method defined by one or more of its parent classes, often the new implementation simply needs to augment the functionality of the existing implementation(s). Rather than completely re-implementing the method, the overriding method procedure can conveniently invoke the procedure that one or more of the parent classes uses to implement that method, then perform additional computation (redefinition) as needed. The parent method call can occur anywhere within the overriding method. (See Example 3 of the SOM IDL tutorial.)

The SOM-generated implementation header file defines the following macros for making parent-method calls from within an overriding method:

<*className*>**_parent_**<*parentClassName*>_<*methodName*>
(for each parent class of the class overriding the method), and

<*className*>**_parents_**<*methodName*>.

For example, given class "Hello" with parents "File" and "Printer" and overriding method **somInit** (the SOM method that initializes each object), the SOM Compiler defines the following macros in the implementation header file for "Hello":

```
Hello_parent_Printer_somInit
Hello_parent_File_somInit
Hello_parents_somInit
```

Each macro takes the same number and type of arguments as <*methodName*>. The <*className*>**_parent_**<*parentClassName*>_<*methodName*> macro invokes the implementation of <*methodName*> inherited from <*parentClassName*>. Hence, using the macro "Hello_parent_File_somInit" invokes File's implementation of **somInit**.

The <*className*>**_parents_**<*methodName*> macro invokes the parent method for *each* parent of the child class that supports <*methodName*>. That is, "Hello_parents_somInit" would invoke File's implementation of **somInit**, followed by Printer's implementation of **somInit**. The <*className*>**_parents_**<*methodName*> macro is redefined in the binding file each time the class interface is modified, so that if a parent class is added or removed from the class definition, or if <*methodName*> is added to one of the existing parents, the macro <*className*>**_parents_**<*methodName*> will be redefined appropriately.

Converting C++ classes to SOM classes

For C++ programmers implementing SOM classes, SOM provides a macro that simplifies the process of converting C++ classes to SOM classes. This macro allows the implementation of one method of a class to invoke another new or overriding method of the same class on the same receiving object by using the following shorthand syntax:

```
_methodName(arg1, arg2, ...)
```

For example, if class *X* introduces or overrides methods *m1* and *m2*, then the C++ implementation of method *m1* can invoke method *m2* on its *somSelf* argument using *_m2(arg, arg2, ...)*, rather than *somSelf–>m2(arg1, arg2, ...)*, as would otherwise be required. (The longer form is also available.) Before the shorthand form in the implementation file is used, the macro **METHOD_MACROS** must be defined (that is, use **#define METHOD_MACROS**) prior to including the .xih file for the class.

Running incremental updates of the implementation template file

Refining the .idl file for a class is typically an iterative process. For example, after running the IDL source file through the SOM Compiler and writing some code in the implementation template file, the class implementor realizes that the IDL class interface needs another method or attribute, a method needs a different parameter, or any such changes.

As mentioned earlier, the SOM Compiler (when run using the **c** or **xc** emitter) assists in this development by reprocessing the .idl file and making *incremental updates* to the current implementation file. This modify-and-update process may in fact be repeated several times before the class declaration becomes final. Importantly, these updates do not disturb existing code for the method procedures. Included in the incremental update are these changes:

- Stub procedures are inserted into the implementation file for any new methods added to the .idl file.
- New comments in the .idl file are transferred to the implementation file, reformatted appropriately.
- If the interface to a method has changed, a new method procedure prototype is placed in the implementation file. As a precaution, however, the old prototype is also preserved within comments. The body of the method procedure is left untouched (as are the method procedures for all methods).
- Similarly left intact are preprocessor directives, data declarations, constant declarations, non-method functions, and additional comments — in essence, everything else in the implementation file.

Some changes to the .idl file are *not* reflected automatically in the implementation file after an incremental update. The class implementor must manually edit the implementation file after changes such as these in the .idl file:

- Changing the name of a class or a method.
- Changing the parents of a class (see also “If you change the parents of a class...” later in this topic).
- Changing a **functionprefix** class modifier statement.
- Changing the content of a **passthru** statement directed to the implementation (.c, .C, or .cpp) file. As previously emphasized, however, **passthru** statements are primarily recommended only for placing #include statements in a binding file (.ih, .xih, .h, or .xh file) used as a header file in the implementation file or in a client program.
- If the class implementor has placed “forward declarations” of the method procedures in the implementation file, those are not updated. Updates occur only for method prototypes that are part of the method procedures themselves.

Considerations to ensure that updates work
To ensure that the SOM Compiler can properly update method procedure prototypes in the implementation file, class implementors should avoid editing changes such as the following:

- A method procedure name should *not* be enclosed in parentheses in the prototype.
- A method procedure name must appear in the first line of the prototype, excluding comments and whitespace. Thus, a newline must *not* be inserted before the procedure name.

Error messages occur while updating an existing implementation file if it contains syntax that is not ANSI C. For example, "old style" method definitions such as the example on the left generate errors:

Invalid "old" syntax	Required ANSI C

`void foo(x)`
`short x;`
`{`
` ...`
`}`
	`void foo(short x) {`
	` ...`
	`}`

Similarly, error messages occur if anything in the .idl file would produce an implementation file that is not syntactically valid for C/C++ (such as nested comments). If update errors occur, either the .idl file or the implementation file may be at fault. One way to track down the problem is to run the implementation file through the C/C++ compiler. Or, move the existing implementation file to another directory, generate a completely new one from the .idl file, and then run *it* through the C/C++ compiler. One of these steps should pinpoint the error, if the compiler is strict ANSI.

Conditional compilation (using #if and #ifdef directives) in the implementation file can be another source of errors, because the SOM Compiler does not invoke the preprocessor (it simply recognizes and ignores those directives). The programmer should be careful when using conditional compilation, to avoid a situation such as shown below; here, with apparently two open braces and only one closing brace, the **c** or **xc** emitter would report an unexpected end-of-file:

Invalid syntax

```
#ifdef FOOBAR
 {
  ...
#else
 {
  ...
#endif
 }
```

Required matching braces

```
#ifdef FOOBAR
 {
  ...
 }
#else
 {
  ...
 }
#endif
```

If you change the parents of a class...
Because the implementation-file emitters never change any existing code within a previously generated implementation file, changing the parents of a class requires extremely careful attention by the programmer. For example, for overridden methods, changing a class's parents may invalidate previous parent-method calls provided by the template, and require the addition of new parent-method calls. Neither of these issues is addressed by the incremental update of previously generated method-procedure templates.

The greatest danger from changing the parents of a class, however, concerns the ancestor-initializer calls provided in the stub procedures for initializer methods. (For further information on ancestor initializer calls, see "Initializing and Uninitializing Objects" later in this chapter.) Unlike parent-method calls, ancestor-initializer calls are not optional — they must be made to all classes specified in a **directinitclasses** modifier, and these calls should always include the parents of the class (the default when no **directinitclasses** modifier is given). When the parents of a class are changed, however, the ancestor-initializer calls (which must be made in a specific order) are not updated.

The easiest way to deal with this problem is to change the method name of the previously generated initializer stub procedure in the implementation template file. Then, the SOM Compiler can correctly generate a completely new initializer stub procedure (while ignoring the renamed procedure). Once this is done, your customization code from the renamed initializer procedure can be "merged" into the newly generated one, after which the renamed initializer procedure can be deleted.

Compiling and linking

After you fill in the method stub procedures, the implementation template file can be compiled and linked with a client program as shown below. In these examples, the environment variable SOMBASE represents the directory in which SOM has been installed.

Note: If you are building an application that uses a combination of C and C++ compiled object modules, then the C++ linker must be used.

For AIX: When the client program (main.c) and the implementation file (hello.c) are written in C:

```
> xlc –I. –I$SOMBASE/include –o hello main.c hello.c \
      –L$SOMBASE/lib –lsomtk
```

When the client program and the implementation file are written in C++:

```
> xlC –I. –I$SOMBASE/include –o hello main.C hello.C \
      –L$SOMBASE/lib –lsomtk
```

For OS/2: When the client program (main.c) and the implementation file (hello.c) are in C:

```
> set LIB=%SOMBASE%\lib;%LIB%
> icc –I. –I%SOMBASE%\include –Fe hello \
      main.c hello.c somtk.lib
```

When the client program and the implementation file are written in C++:

```
> set LIB=%SOMBASE%\lib;%LIB%
> icc –I. –I%SOMBASE%\include –Fe hello \
      main.cpp hello.cpp somtk.lib
```

For Windows: When the client program (main.c) and the implementation file (hello.c) are in C:

```
> cl –AL –Zp1 –I. –I%SOMBASE%\include –Fehello \
      main.c hello.c llibcew.lib libw.lib somtk.lib main.def
```

When the client program and the implementation file are written in C++:

```
> cl –AL –Zp1 –I. –I%SOMBASE%\include –Fehello \
      main.cpp hello.cpp llibcew.lib libw.lib somtk.lib main.def
```

Windows compiler notes: The SOM Compiler and runtime libraries under Windows only provide support for Large Memory Model systems. The "–Oi" compile flag should be used whenever a SOM Windows program contains or calls any functions that return floats or doubles. The "–F" option can be used to set the stack size if STACKSIZE is not specified in a .def file. Additionally, for this manual, compile and link commands are demonstrated using the Visual C++ compiler. Sample Makefiles shipped with SOMobjects For Windows also show how to build with other compilers.

For all users: If the class definition (in the .idl file) changes, run the SOM Compiler again. This will generate new header files, and it will update the implementation file to include any:

- New comments,
- Stub procedures for any new methods, and
- Revised method procedure prototypes for methods whose signatures have been changed in the .idl file.

After rerunning the SOM Compiler, add to the implementation file the code for any newly added method procedures, and recompile the implementation file with the client program.

5.5 Initializing and Uninitializing Objects

This section discusses the initialization and uninitialization of SOM objects. Subsequent topics introduce the methods and capabilities that the SOMobjects Developer Toolkit provides to facilitate this.

Object creation is the act that enables the execution of methods on an object. In SOM, this means storing a pointer to a method table into a word of memory. This single act converts raw memory into an (uninitialized) SOM object that starts at the location of the method table pointer.

Object initialization, on the other hand, is a separate activity from object creation in SOM. Initialization is a capability supported by certain methods available on an object. An object's class determines the implementation of the methods available on the object, and thus determines its initialization behavior.

The instance variables encapsulated by a newly created object must be brought into a consistent state before the object can be used. This is the purpose of *initialization methods*. Because, in general, every ancestor of an object's class contributes instance data to an object, it is appropriate that each of these ancestors contribute to the initialization of the object.

SOM thus recognizes *initializers* as a special kind of method. One advantage of this approach is that special metaclasses are not required for defining constructors (class methods) that take arguments. Furthermore, a class can define multiple initializer methods, thus enabling its different objects to be initialized supporting different characteristics or capabilities. This results in simpler designs and more efficient programs.

The SOMobjects Toolkit provides an overall framework that class designers can easily exploit in order to implement default or customized initialization of SOM objects. This framework is fully supported by the SOM Toolkit emitters that produce the implementation template file. The following sections describe the declaration, implementation, and use of initializer (and uninitializer) methods.

Important: All code written prior to SOMobjects Release 2.1 using documented guidelines for the earlier initialization approach based on the **somInit** method (as well as all existing class binaries) continues to be fully supported and useful.

Initializer methods

As noted above, in the SOMobjects Toolkit each ancestor of an object contributes to the initialization of that object. Initialization of an object involves a chain of ancestor-method calls that, by default, are automatically determined by the SOM Compiler emitters. The SOMobjects framework for initialization of objects is based on the following approach:

1. SOMobjects recognizes *initializers* as a special kind of method, and supports a special mechanism for ordering the execution of ancestor-initializer method procedures. The **SOMObject** class introduces an initializer method, **somDefaultInit**, that uses this execution mechanism.
2. The SOM Compiler's emitters provide special support for methods that are *declared* as initializers in the .idl file. To supplement the **somDefaultInit** method, SOM class designers can also declare additional initializers in their own classes.

Two SOM IDL modifiers are provided for declaring initializer methods and controlling their execution, **init** and **directinitclasses**:

- The **init** modifier is required in order to designate a given method is an initializer; that is, to indicate that the method both uses and supports the object-initialization protocol described here.
- The **directinitclasses** modifier can be used to control the order of execution of initializer method procedures provided by the different ancestors of the class of an object.

For full definitions of **init** and **directinitclasses**, see the topic “Modifier statements” in Chapter 4,“SOM IDL and the SOM Compiler.”

Every SOM class has a list that defines (in sequential order) the ancestor classes whose initializer method procedures the class should invoke. If a class’s IDL does not specify an explicit **directinitclasses** modifier, the default for this list is simply the class’s parents — in left-to-right order.

Using the **directinitclasses** list and the actual run-time class hierarchy above itself, each class inherits from **SOMClass** the ability to create a data structure of type **somInitCtrl**. This structure is used to control the execution of initializers. Moreover, it represents a particular visit-ordering that reaches each class in the transitive closure of **directinitclasses** exactly once. To initialize a given object, this visit-ordering occurs as follows: While recursively visiting each ancestor class whose initializer method procedure should be run, SOMobjects first runs the initializer method procedures of all of that class’s **directinitclasses** if they have not already been run by another class’s initializers, with ancestor classes always taken in left-to-right order.

For example, Figure 10 shows an inheritance hierarchy along with the ordering produced when an instance of the class numbered **7** is initialized, assuming that each class simply uses its parents as its **directinitclasses**. Note that the class numbered **3** is at the top of a diamond.

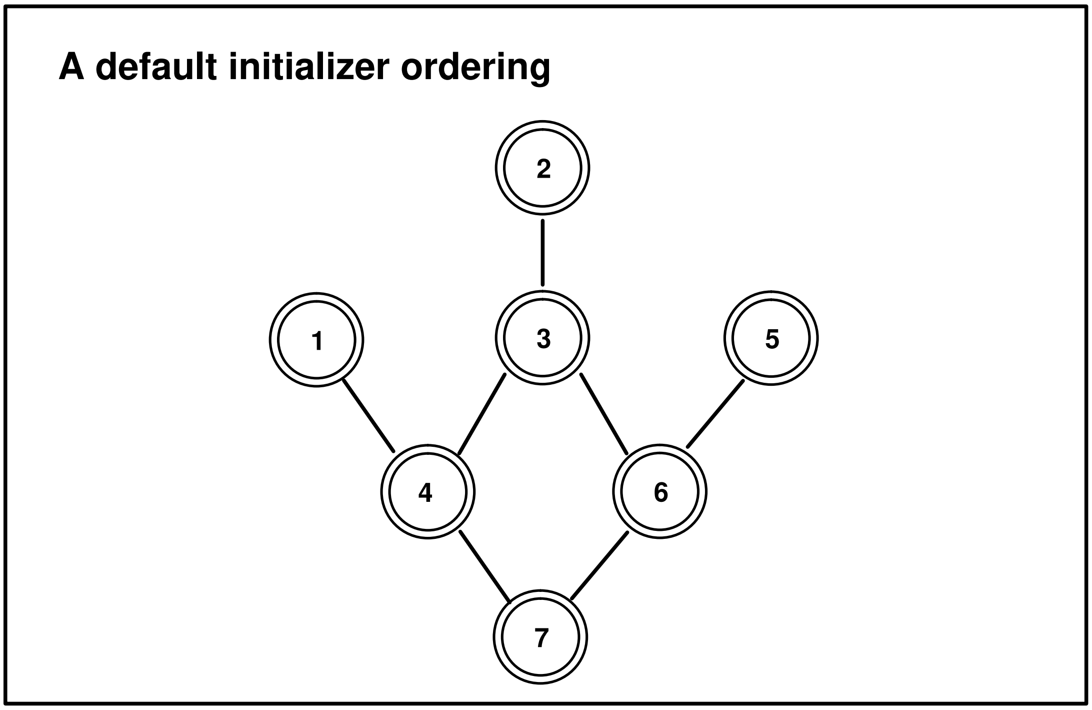


Figure 10. A default initializer ordering of a class’s inheritance hierarchy.

In this example, the **somInitCtrl** data structure for class **7** is what tells node **6** in Figure 10 not to invoke node **3**’s initializer code (because it has already been executed). The code that deals with the **somInitCtrl** data structure is generated automatically within the implementation bindings for a class, and need not concern a class implementor.

As illustrated by this example, when an instance of a given class (or some descendant class) is initialized, only one of the given class’s initializers will be executed, and this will happen exactly once (under control of the ordering determined by the class of the object being initialized).

The **somInitCtrl** structure solves a problem originally created by the addition of multiple inheritance to SOMobjects 2.0. With multiple inheritance, any class can appear at the top of a multiple inheritance diamond. Previously, whenever this happened, the class could easily receive multiple initialization calls. In the current version of the SOMobjects Toolkit, however, the **somInitCtrl** structure prevents this from happening.

Declaring new initializers in SOM IDL

When defining SOMobjects classes, programmers can easily declare and implement new initializers. Classes can have as many initializers as desired, and subclassers can invoke whichever of these they want. When introducing new initializers, developers must adhere to the following rules:

- All initializer methods take a **somInitCtrl** data structure as an initial **inout** parameter (its type is defined in the SOMobjects header file somapi.h), and
- All initializers return **void**.

Accordingly, the **somDefaultInit** initializer introduced by **SOMObject** takes a **somInitCtrl** structure as its (only) argument, and returns **void**. Here is the IDL syntax for this method, as declared in somobj.idl:

void somDefaultInit (inout somInitCtrl *ctrl***);**

When introducing a new initializer, it is also necessary to specify the **init** modifier in the **implementation** section. The **init** modifier is what tells emitters that the new method is actually an initializer, so the method can be properly supported from the language bindings. As described below, this support includes the generation of special initializer stub procedures in the implementation template file, as well as bindings containing ancestor-initialization macros and object constructors that invoke the class implementor's new initializers.

It is a good idea to begin the names of initializer methods with the name of the class (or some other string that can be unique for the class). This is important because all initializers available on a class must be newly introduced by that class (that is, you cannot override initializers — except for **somDefaultInit**). Using a class-unique name means that subclasses will not be unnecessarily constrained in their choice of initializer names.

Here are two classes that introduce new initializers:

```
interface Example1 : SOMObject
{
   void Example1_withName (inout somInitCtrl ctrl, in string name);
   void Example1_withSize (inout somInitCtrl ctrl, in long size);
   void Example1_withNandS(inout somInitCtrl ctrl, in string name,
                                                   in long size);
        implementation {
                releaseorder: Example1_withName,
                              Example1_withSize,
                              Example1_withNandS;
                somDefaultInit: override, init;
                somDestruct: override;
                Example1_withName: init;
                Example1_withSize: init;
                Example1_withNandS: init;
        };
};
```

```
interface Example2 : Example1
{
   void Example2_withName(inout somInitCtrl ctrl, in string name);
   void Example2_withSize(inout somInitCtrl ctrl, in long size);
        implementation {
                releaseorder: Example2_withName,
                              Example2_withSize;
                somDefaultInit: override, init;
                somDestruct: override;
                Example2_withName: init;
                Example2_withSize: init;
        };
};
```

Here, interface “Example1” declares three new initializers. Notice the use of **inout somInitCtrl** as the first argument of each initializer, and also note that the **init** modifier is used in the **implementation** section. These two things are required to declare initializers. Any number of initializers can be declared by a class. “Example2” declares two initializers.

“Example1” and “Example2” both override the **somDefaultInit** initializer. This initializer method is introduced by **SOMObject** and is special for two reasons: First, **somDefaultInit** is the *only* initializer that can be overridden. And, second, SOMobjects arranges that this initializer will always be available on any class (as further explained below).

Historically in the SOMobjects Toolkit, object-initialization methods by default have invoked the **somInit** method, which class implementors could override to customize initialization as appropriate. SOMobjects continues to support this approach, so that existing code (and class binaries) will execute correctly. However, the **somDefaultInit** method is now the preferred form of initialization because it offers greatly improved efficiency.

Even if no specialized initialization is needed for a class, you should still **override** the **somDefaultInit** method in the interest of efficiency. If you do not override **somDefaultInit**, then a generic (and therefore less efficient) **somDefaultInit** method procedure will be used for your class. This generic method procedure first invokes **somDefaultInit** on the appropriate ancestor classes. Then (for consistency with earlier versions of SOMobjects), it checks to determine if the class overrides **somInit** and, if so, calls any customized **somInit** code provided by the class.

When you override **somDefaultInit**, the emitter’s implementation template file will include a stub procedure similar to those used for other initializers, and you can fill it in as appropriate (or simply leave it as is). Default initialization for your class will then run much faster than with the generic method procedure. Examples of initializer stub procedures (and customizations) are given below.

In summary, the initializers available for any class of objects are **somDefaultInit** (which you should always **override**) plus any new initializers explicitly declared by the class designer. Thus, “Example1” objects may be initialized using any of four different initializers (the three that are explicitly declared, plus **somDefaultInit**). Likewise, there are three initializers for the “Example2” objects. Some examples of using initializers are provided below.

Considerations re: ‘somInit’ initialization from earlier SOM releases

To re-emphasize: All code written prior to SOMobjects Release 2.1 using documented guidelines for the earlier initialization approach based on the **somInit** method (as well as all existing class binaries) continues to be fully supported and useful.

Prior to SOMobjects 2.1, initialization was done with initializer methods that would simply “chain” parent-method calls upward, thereby allowing the execution of initializer method procedures contributed by all ancestors of an object’s class. This chaining of initializer calls was not supported in any special way by the SOM API. Parent-method calls are simply one of the possible idioms available to users of OOP in SOM, easily available to a SOM class designer as a result of the support provided by the SOMobjects Toolkit emitters for parent-method calls.

So, SOM did not constrain initialization to be done in any particular way or require the use of any particular ordering of the method procedures of ancestor classes. But, SOM did provide an overall framework that class designers could easily utilize in order to implement default initialization of SOM objects. This framework is provided by the **somInit** object-initialization method introduced by the **SOMObject** class and supported by the SOM Toolkit emitters. The emitters create an implementation template file with stub procedures for overridden methods that automatically chain parent-method calls upward through parent classes. Many of the class methods that perform object creation called **somInit** automatically. [Note: These will now call **somDefaultInit**, which in turn calls **somInit** for legacy code, as described in the previous topic.]

Because it takes no arguments, **somInit** best served the purpose of a default initializer. SOM programmers also had the option of introducing additional "non-default" initialization methods that took arguments. In addition, by using metaclasses, they could introduce new class methods as object constructors that first create an object (generally using **somNewNoInit**) and then invoke some non-default initializer on the new object.

For a number of reasons, the **somInit** framework has been augmented by recognizing *initializers* as a special kind of method in SOMobjects. One advantage of this approach is that special metaclasses are no longer required for defining constructors that take arguments. Instead, because the **init** modifier identifies initializers, usage-binding emitters can now provide these constructors. This results in simpler designs and more efficient programs.

Although **somDefaultInit** replaces **somInit** as the no-argument initializer used for SOM objects, all previous use of **somInit** is still supported by the SOMobjects Developers Toolkit on AIX, OS/2 and 16-bit Windows. You may continue to use **somInit** on these systems if you like, although this is somewhat less efficient than using **somDefaultInit**.

However, you *cannot* use both methods. In particular, if a class overrides both **somDefaultInit** and **somInit**, its **somInit** code will never be executed. It is recommended that you always override **somDefaultInit** for object initialization. For one thing, it is likely that when SOMobjects is ported to new systems, **somInit** (and **somUninit**) may not be supported on those systems. Thus, code written using these (obsolete) methods will be less portable.

Implementing initializers

When new initializers are introduced by a class, as in the preceding examples, the implementation template file generated by the SOM Toolkit C and C++ emitters automatically contains an appropriate stub procedure for each initializer method, for the class implementor's use. The body of an initializer stub procedure consists of two main sections:

- The first section performs calls to ancestors of the class to invoke their initializers.
- The second section is used by the programmer to perform any "local" initializations appropriate to the instance data of the class being defined.

In the first section, by default, the parents of the new class are the ancestors whose initializers are called. When something else is desired, the IDL **directinitclasses** modifier can be used to explicitly designate the ancestors whose initializer methods should be invoked by a new class's initializers.

Important: Under no circumstances can the number or the ordering of ancestor initializer calls in the first section of an initializer stub procedure be changed. The control masks used by initializers are based on these orderings. (If you want to change the number or ordering of ancestor initializer calls, you must use the **directinitclasses** modifier.) The ancestor initializer calls themselves can be modified as described below.

Each call to an ancestor initializer is made using a special macro (much like a parent call) that is defined for this purpose within the implementation bindings. These macros are defined for all possible ancestor initialization calls. Initially, an initializer stub procedure invokes the default ancestor initializers provided by **somDefaultInit**. However, a class implementor can replace any of these calls with a different initializer call, as long as it calls the same ancestor (see the example in the next topic). Non-default initializer calls generally take other arguments in addition to the control argument.

In the second section of an initializer stub procedure, the programmer provides any class-specific code that may be needed for initialization. For example, the “Example2_withName” stub procedure is shown below. As with all stub procedures produced by the SOMobjects implementation-template emitters, this code requires no modification to run correctly.

```
SOM_Scope void SOMLINK Example2_withName(Example2 *somSelf,
                                         Environment *ev,
                                         somInitCtrl* ctrl,
                                         string name)
{
    Example2Data *somThis; /* set by BeginInitializer */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    Example2MethodDebug("Example2","withName");

    /*
     * first section -- calls to ancestor initializers
     */
    Example2_BeginInitializer_Example2_withName;
    Example2_Init_Example1_somDefaultInit(somSelf, ctrl);

    /*
     * second section -- local Example2 initialization code
     */
}
```

In this example, notice that the “Example2_withName” initializer is an IDL callstyle method, so it receives an **Environment** argument. In contrast, **somDefaultInit** is introduced by the **SOMObject** class (so it has an OIDL callstyle initializer, without an environment).

Important: If a class is defined where multiple initializers have exactly the same signature, then the C++ usage bindings will not be able to differentiate among them. That is, if there are multiple initializers defined with environment and long arguments, for example, then C++ clients would *not* be able to make a call using only the class name and arguments, such as:

```
new Example2(env, 123);
```

Rather, C++ users would be forced to first invoke the **somNewNoInit** method on the class to create an uninitialized object, and then separately invoke the desired initializer method on the object. This call would pass a zero for the control argument, in addition to passing values for the other arguments. For further discussion of client usage, see “Using initializers when creating new objects” later in this chapter.

Selecting non-default ancestor initializer calls

Often, it will be appropriate (in the first section of an initializer stub procedure) to change the invocation of an ancestor’s **somDefaultInit** initializer to some other initializer available on the same class. The rule for making this change is simple: Replace **somDefaultInit** with the name of the desired ancestor initializer, and add any new arguments that are required by the replacement initializer. Important: Under no circumstances can you change anything else in the first section.

This example shows how to change an ancestor-initializer call correctly. Since there is a known “Example1_withName” initializer, the following default ancestor-initializer call (produced within the stub procedure for “Example2_withName”) can be changed from

```
Example2_Init_Example1_somDefaultInit(somSelf, ctrl);
```

to

```
Example2_Init_Example1_Example1_withName(somSelf, ev, ctrl, name);
```

Notice that the revised ancestor-initializer call includes arguments for an **Environment** and a name, as defined by the "Example1_withname" initializer.

Using initializers when creating new objects

There are several ways that client programs can take advantage of the **somDefaultInit** object initialization. If desired, clients can use the SOM API directly (rather than taking advantage of the usage bindings). Also, the general object constructor, **somNew**, can always be invoked on a class to create and initialize objects. This call creates a new object and then invokes **somDefaultInit** on it.

To use the SOM API directly, the client code should first invoke the **somNewNoInit** method on the desired class object to create a new, uninitialized object. Then, the desired initializer is invoked on the new object, passing a null (that is, 0) control argument in addition to whatever other arguments may be required by the initializer. For example:

```
/* first make sure the Example2 class object exists */
Example2NewClass(Example2_MajorVersion, Example2_MinorVersion);

/* then create a new, uninitialized Example2 object */
myObject = _somNewNoInit(_Example2);

/* then initialize it with the desired initializer */
Example2_withName(myObject, env, 0, "MyName");
```

Usage bindings hide the details associated with initializer use in various ways and make calls more convenient for the client. For example, the C usage bindings for any given class already provide a convenience macro, <*className*>**New**, that first assures existence of the class object, and then calls **somNew** on it to create and initialize a new object. As explained above, **somNew** will use **somDefaultInit** to initialize the new object.

Also, the C usage bindings provide object-construction macros that use **somNewNoInit** and then invoke non-default initializers. These macros are named using the form <*className*>**New_**<*initializerName*>. For example, the C usage bindings for "Example2" allow using the following expression to create, initialize, and return a new "Example2" object:

```
Example2New_Example2_withName(env, "AnyName");
```

In the C++ bindings, initializers are represented as overloaded C++ constructors. As a result, there is no need to specify the name of the initializer method. For example, using the C++ bindings, the following expressions could be used to create a new "Example2" object:

```
new Example2;                        // will use somDefaultInit
new Example2();                      // will use somDefaultInit
new Example2(env,"A.B.Normal"); // will use Example2_withName
new Example2(env,123);               // will use Example2_withSize
```

Observe that if multiple initializers in a class have exactly the same signatures, the C++ usage bindings would be unable to differentiate among the calls, if made using the forms illustrated above. In this case, a client could use **somNewNoInit** first, and then invoke the specific initializer, as described in the preceding paragraphs.

Uninitialization

An object should always be uninitialized before its storage is freed. This is important because it also allows releasing resources and freeing storage not contained within the body of the object. SOMobjects handles uninitialization in much the same way as for initializers: An uninitializer takes a control argument and is supported with stub procedures in the implementation template file in a manner similar to initializers.

Only a single uninitialization method is needed, so **SOMObject** introduces the method that provides this function: **somDestruct**. As with the default initializer method, a class designer who requires nothing special in the way of uninitialization need not be concerned about modifying the default **somDestruct** method procedure. However, your code will execute faster if the .idl file overrides **somDestruct** so that a non-generic stub-procedure code can be provided for the class. Note that **somDestruct** was overridden by "Example1" and "Example2" above. No specific IDL modifiers other than **override** are required for this.

Like an initializer template, the stub procedure for **somDestruct** consists of two sections: The first section is used by the programmer for performing any "local" uninitialization that may be required. The second section (which consists of a single **EndDestructor** macro invocation) invokes **somDestruct** on ancestors. The second section must *not* be modified or removed by the programmer. It must be the final statement executed in the destructor.

Using 'somDestruct'

It is rarely necessary to invoke the **somDestruct** method explicitly. This is because object uninitialization is normally done just before freeing an object's storage, and the mechanisms provided by SOMobjects for this purpose will automatically invoke **somDestruct**. For example, if an object were created using **somNew** or **somNewNoInit**, or by using a convenience macro provided by the C language bindings, then the **somFree** method can be invoked on the object to delete the object. This automatically calls **somDestruct** before freeing storage.

C++ users can simply use the **delete** operator provided by the C++ bindings. This destructor calls **somDestruct** before the C++ **delete** operator frees the object's storage.

On the other hand, if an object is initially created by allocating memory in some special way and subsequently some **somRenew** methods are used, **somFree** (or C++ **delete**) is probably not appropriate. Thus, the **somDestruct** method should be explicitly called to uninitialize the object before freeing memory.

A complete example

The following example illustrates the implementation and use of initializers and destructors from the C++ bindings. The first part shows the IDL for three classes with initializers. For variety, some of the classes use callstyle OIDL and others use callstyle IDL.

```
#include <somobj.idl>

interface A : SOMObject {
        readonly attribute long a;
        implementation {
                releaseorder: _get_a;
                functionprefix = A;
                somDefaultInit: override, init;
                somDestruct: override;
                somPrintSelf: override;
        };
};


interface B : SOMObject {
        readonly attribute long b;
        void BwithInitialValue(inout somInitCtrl ctrl,
                               in long initialValue);
        implementation {
                callstyle = OIDL;
                releaseorder: _get_b, BwithInitialValue;
                functionprefix = B;
                BwithInitialValue: init;
                somDefaultInit: override, init;
                somDestruct: override;
                somPrintSelf: override;
        };
};


interface C : A, B      {
        readonly attribute long c;
        void CwithInitialValue(inout somInitCtrl ctrl,
                               in long initialValue);
        void CwithInitialString(inout somInitCtrl ctrl,
                                in string initialString);
        implementation {
                releaseorder: _get_c, CwithInitialString,
                              CwithInitialValue;
                functionprefix = C;
                CwithInitialString: init;
                CwithInitialValue: init;
                somDefaultInit: override, init;
                somDestruct: override;
                somPrintSelf: override;
        };
};
```

Implementation code

Based on the foregoing class definitions, the next example illustrates several important aspects of initializers. The following code is a completed implementation template and an example client program for the preceding classes. Code added to the original template is given in bold.

```
/*
 *  This file generated by the SOM Compiler and Emitter Framework.
 *  Generated using:
 *      SOM Emitter emitxtm.dll: 2.22
 */

#define SOM_Module_ctorfullexample_Source
#define VARIABLE_MACROS
#define METHOD_MACROS
#include <ctorFullExample.xih>
#include <stdio.h>

SOM_Scope void SOMLINK AsomDefaultInit(A *somSelf,
                                       somInitCtrl* ctrl)
{
    AData *somThis; /* set by BeginInitializer */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    AMethodDebug("A","somDefaultInit");

    A_BeginInitializer_somDefaultInit;
    A_Init_SOMObject_somDefaultInit(somSelf, ctrl);
    /*
     * local A initialization code added by programmer
     */
    _a = 1;
}

SOM_Scope void SOMLINK AsomDestruct(A *somSelf, octet doFree,
                                    somDestructCtrl* ctrl)
{
    AData *somThis; /* set by BeginDestructor */
    somDestructCtrl globalCtrl;
    somBooleanVector myMask;
    AMethodDebug("A","somDestruct");
    A_BeginDestructor;

    /*
     * local A deinitialization code added by programmer
     */
    A_EndDestructor;
}

SOM_Scope SOMObject*  SOMLINK AsomPrintSelf(A *somSelf)
{
    AData *somThis = AGetData(somSelf);
    AMethodDebug("A","somPrintSelf");

    somPrintf("{an instance of %s at location %X with (a=%d)}\n",
              _somGetClassName(),somSelf,__get_a((Environment*)0));
    return (SOMObject*)((void*)somSelf);
}
```

```
SOM_Scope void SOMLINK BBwithInitialValue(B *somSelf,
                                          somInitCtrl* ctrl,
                                          long initialValue)
{
    BData *somThis; /* set by BeginInitializer */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    BMethodDebug("B","BwithInitialValue");

    B_BeginInitializer_withInitialValue;
    B_Init_SOMObject_somDefaultInit(somSelf, ctrl);

    /*
     * local B initialization code added by programmer
     */
    _b = initialValue;
}

SOM_Scope void SOMLINK BsomDefaultInit(B *somSelf,
                                       somInitCtrl* ctrl)
{
    BData *somThis; /* set by BeginInitializer */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    BMethodDebug("B","somDefaultInit");

    B_BeginInitializer_somDefaultInit;
    B_Init_SOMObject_somDefaultInit(somSelf, ctrl);

    /*
     * local B initialization code added by programmer
     */
    _b = 2;
}

SOM_Scope void SOMLINK BsomDestruct(B *somSelf, octet doFree,
                                    somDestructCtrl* ctrl)
{
    BData *somThis; /* set by BeginDestructor */
    somDestructCtrl globalCtrl;
    somBooleanVector myMask;
    BMethodDebug("B","somDestruct");
    B_BeginDestructor;

    /*
     * local B deinitialization code added by programmer
     */

    B_EndDestructor;
}

SOM_Scope SOMObject*  SOMLINK BsomPrintSelf(B *somSelf)
{
    BData *somThis = BGetData(somSelf);
    BMethodDebug("B","somPrintSelf");

    printf("{an instance of %s at location %X with (b=%d)}\n",
           _somGetClassName(),somSelf,__get_b());
    return (SOMObject*)((void*)somSelf);
}
```

Note: The following initializer for a C object accepts a string as an argument, converts this to an integer, and uses this for ancestor initialization of “B.” This illustrates how a default ancestor initializer call is replaced with a non-default ancestor initializer call.

```
SOM_Scope void SOMLINK CCwithInitialString(C *somSelf,
                                           Environment *ev,
                                           somInitCtrl* ctrl,
                                           string initialString)
{
    CData *somThis; /* set by BeginInitializer */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    CMethodDebug("C","CwithInitialString");

    C_BeginInitializer_withInitialString;
    C_Init_A_somDefaultInit(somSelf, ctrl);
    C_Init_B_BwithInitialValue(somSelf, ctrl,
                               atoi(initialString)-11);

    /*
     * local C initialization code added by programmer
     */
    _c = atoi(initialString);
}

SOM_Scope void SOMLINK CsomDefaultInit(C *somSelf,
                                       somInitCtrl* ctrl)
{
    CData *somThis; /* set by BeginInitializer */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    CMethodDebug("C","somDefaultInit");

    C_BeginInitializer_somDefaultInit;
    C_Init_A_somDefaultInit(somSelf, ctrl);
    C_Init_B_somDefaultInit(somSelf, ctrl);

    /*
     * local C initialization code added by programmer
     */
    _c = 3;
}

SOM_Scope void SOMLINK CsomDestruct(C *somSelf, octet doFree,
                                    somDestructCtrl* ctrl)
{
    CData *somThis; /* set by BeginDestructor */
    somDestructCtrl globalCtrl;
    somBooleanVector myMask;
    CMethodDebug("C","somDestruct");
    C_BeginDestructor;

    /*
     * local C deinitialization code added by programmer
     */

    C_EndDestructor;
}
```

```
SOM_Scope SOMObject*  SOMLINK CsomPrintSelf(C *somSelf)
{
    CData *somThis = CGetData(somSelf);
    CMethodDebug("C","somPrintSelf");

    printf("{an instance of %s at location %X with"
           " (a=%d, b=%d, c=%d)}\n",
        _somGetClassName(),somSelf,
        __get_a((Environment*)0),
        __get_b(),
        __get_c((Environment*)0));
    return (SOMObject*)((void*)somSelf);
}

SOM_Scope void SOMLINK CCwithInitialValue(C *somSelf,
                                          Environment *ev,
                                          somInitCtrl* ctrl,
                                          long initialValue)
{
    CData *somThis; /* set by BeginInitializer */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    CMethodDebug("C","CwithInitialValue");

    C_BeginInitializer_withInitialValue;
    C_Init_A_somDefaultInit(somSelf, ctrl);
    C_Init_B_BwithInitialValue(somSelf, ctrl, initialValue-11);

    /*
     * local C initialization code added by programmer
     */
    _c = initialValue;
}
```

Here is a C++ program that creates instances of "A", "B", and "C" using the initializers defined above.

```
main()
{
    SOM_TraceLevel = 1;

    A *a = new A;
    a->somPrintSelf();
    delete a;
    printf("\n");

    B *b = new B();
    b->somPrintSelf();
    delete b;
    printf("\n");

    b = new B(22);
    b->somPrintSelf();
    delete b;
    printf("\n");

    C *c = new C;
    c->somPrintSelf();
    delete c;
    printf("\n");
```

```
    c = new C((Environment*)0, 44);
    c->somPrintSelf();
    delete c;
    printf("\n");

    c = new C((Environment*)0, "66");
    c->somPrintSelf();
    delete c;
}
```

The output from the preceding program is as follows:

```
"ctorFullExample.C": 18:        In A:somDefaultInit
"ctorFullExample.C": 48:        In A:somPrintSelf
"./ctorFullExample.xih": 292:   In A:A_get_a
{an instance of A at location 20063C38 with (a=1)}
"ctorFullExample.C": 35:        In A:somDestruct

"ctorFullExample.C": 79:        In B:somDefaultInit
"ctorFullExample.C": 110:       In B:somPrintSelf
"./ctorFullExample.xih": 655:   In B:B_get_b
{an instance of B at location 20064578 with (b=2)}
"ctorFullExample.C": 97:        In B:somDestruct

"ctorFullExample.C": 62:        In B:BwithInitialValue
"ctorFullExample.C": 110:       In B:somPrintSelf
"./ctorFullExample.xih": 655:   In B:B_get_b
{an instance of B at location 20064578 with (b=22)}
"ctorFullExample.C": 97:        In B:somDestruct

"ctorFullExample.C": 150:       In C:somDefaultInit
"ctorFullExample.C": 18:        In A:somDefaultInit
"ctorFullExample.C": 79:        In B:somDefaultInit
"ctorFullExample.C": 182:       In C:somPrintSelf
"./ctorFullExample.xih": 292:   In A:A_get_a
"./ctorFullExample.xih": 655:   In B:B_get_b
"./ctorFullExample.xih": 1104:  In C:C_get_c
{an instance of C at location 20065448 with (a=1, b=2, c=3)}
"ctorFullExample.C": 169:       In C:somDestruct
"ctorFullExample.C": 35:        In A:somDestruct
"ctorFullExample.C": 97:        In B:somDestruct

"ctorFullExample.C": 196:       In C:CwithInitialValue
"ctorFullExample.C": 18:        In A:somDefaultInit
"ctorFullExample.C": 62:        In B:BwithInitialValue
"ctorFullExample.C": 182:       In C:somPrintSelf
"./ctorFullExample.xih": 292:   In A:A_get_a
"./ctorFullExample.xih": 655:   In B:B_get_b
"./ctorFullExample.xih": 1104:  In C:C_get_c
{an instance of C at location 20065448 with (a=1, b=33, c=44)}
"ctorFullExample.C": 169:       In C:somDestruct
"ctorFullExample.C": 35:        In A:somDestruct
"ctorFullExample.C": 97:        In B:somDestruct
```

```
"ctorFullExample.C": 132:       In C:CwithInitialString
"ctorFullExample.C": 18:        In A:somDefaultInit
"ctorFullExample.C": 62:        In B:BwithInitialValue
"ctorFullExample.C": 182:       In C:somPrintSelf
"./ctorFullExample.xih": 292:   In A:A_get_a
"./ctorFullExample.xih": 655:   In B:B_get_b
"./ctorFullExample.xih": 1104:  In C:C_get_c
{an instance of C at location 20065448 with (a=1, b=55, c=66)}
"ctorFullExample.C": 169:       In C:somDestruct
"ctorFullExample.C": 35:        In A:somDestruct
"ctorFullExample.C": 97:        In B:somDestruct
```

Customizing the initialization of class objects

As described previously, the **somDefaultInit** method can be overridden to customize the initialization of objects. Because classes are objects, **somDefaultInit** is also invoked on classes when they are first created (generally by invoking the **somNew** method on a metaclass). For a class object, however, **somDefaultInit** normally just sets the name of the class to "unknown," after which the **somInitMIClass** method must be used for the major portion of class initialization. Of course, metaclasses can override **somDefaultInit** to initialize introduced class variables that require no arguments for their initialization.

Note: Because **somNew** does not call **somInitMIClass**, class objects returned from invocations of **somNew** on a metaclass are not yet useful class objects.

The **somInitMIClass** method (introduced by **SOMClass**) is invoked on a new class object using arguments to indicate the class name and the parent classes from which inheritance is desired (among other arguments). This invocation is made by whatever routine is used to initialize the class. (For SOM classes using the C or C++ implementation bindings, this is handled by the **somBuildClass** procedure, which is called by the implementation bindings automatically.) The **somInitMIClass** method is often overridden by a metaclass to influence initialization of new classes in some way. Typically, the overriding procedure begins by making parent method calls, and then performs additional actions thereafter.

However, without use of the Cooperative Metaclass to guarantee correct operation of SOMobjects in general, none of the methods introduced by **SOMClass** should be overridden. As a result, customizing the initialization of class objects (other than through overriding **somDefaultInit** for initialization of class variables) is not recommended in SOMobjects 2.1. Users whose applications require this should request access to the experimental Cooperative Metaclass used to implement the SOMobjects Metaclass Framework. But, metaclasses implemented using the experimental Cooperative Metaclass may require reprogramming when SOMobjects introduces an officially supported Cooperative Metaclass.

5.6 Creating a SOM Class Library

One of the principal advantages of SOM is that it makes "black box" (or binary) reusability possible. Consequently, SOM classes are frequently packaged and distributed as class libraries. A *class library* holds the actual implementation of one or more classes and can be dynamically loaded and unloaded as needed by applications. Importantly, class libraries can also be replaced independently of the applications that use them and, provided that the class implementor observes simple SOM guidelines for preserving binary compatibility, can evolve and expand over time.

General guidelines for class library designers

One of the most important features of SOM is that it allows you to build and distribute class libraries in binary form. Because there is no "fragile base class" problem in SOM, client programs that use your libraries (by subclassing your classes or by invoking the methods in your classes) will not need to be recompiled if you later produce a subsequent version of the library, provided you adhere to some simple restrictions.

1. You should always maintain the syntax and the semantics of your existing interfaces. This means that you cannot take away any exposed capabilities, nor add or remove arguments for any of your public methods.
2. Always maintain the **releaseorder** list, so that it never changes except for additions to the end. The **releaseorder** should contain all of your public methods, the one or two methods that correspond to each public attribute, and a placeholder for each private method (or private attribute method).
3. Assign a higher **minorversion** number for each subsequent release of a class, so that client programmers can determine whether a new feature is present or not. Change the **majorversion** number only when you deliberately wish to break binary compatibility. (See the topic "Modifier statements" in Chapter 4, "SOM IDL and the SOM Compiler" for explanations of the **releaseorder, minorversion** and **majorversion** modifiers.)
4. Under Windows, you should avoid the use of methods or attributes that return structures. In the DOS/Windows environment, there is no universally agreed upon calling convention for returning structures that is observed by all popular language compilers. Instead, define attributes that return pointers to structures, or define methods that take an **out** parameter for passing a structure back to the caller.

   Note that you can always avoid this problem in classes of your own design. However, some of the attributes and methods in the frameworks that come with the SOMobjects Toolkit *do* return structures. Many of these are dictated by the OMG CORBA standard, and could not be avoided. For each of these methods two common calling conventions have been implemented:

   - Microsoft convention, where the address of the structure is returned in AX:DX, and
   - Borland convention, where the caller provides a hidden first argument to receive a copy of the returned structure.

   No action is needed on your part to use the Microsoft convention. To use the Borland convention, you should include the file BCCSTRUC.H following any other "includes" of SOM headers.
5. With each new release of your class library, you have significant degrees of freedom to change much of the implementation detail. You can add to or reorganize your instance variables, add new public or private methods, inject new base classes into your class hierarchies, change metaclasses to more derived ones, and relocate the implementation

of methods upward in your class hierarchies. Provided you always retain the same capabilities and semantics that were present in your first release, none of these changes will break the client programs that use your libraries.

Types of class libraries

Since class libraries are not programs, users cannot execute them directly. To enable users to make direct use of your classes, you must also provide one or more programs that create the classes and objects that the user will need. This section describes how to package your classes in a SOM class library and what you must do to make the contents of the library accessible to other programs.

On AIX, class libraries are actually produced as AIX shared libraries, whereas on OS/2 or Windows they appear as dynamically-linked libraries (or DLLs). The term "DLL" is sometimes used to refer to either an AIX, an OS/2, or a Windows class library, and (by convention only) the file suffix ".dll" is used for SOM class libraries on all platforms.

A program can use a class library containing a given class or classes in one of two ways:

1. If the programmer employs the SOM bindings to instantiate the class and invoke its methods, the resulting client program contains static references to the class. The operating system will automatically resolve those references when the program is loaded, by also loading the appropriate class library.
2. If the programmer uses only the dynamic SOM mechanisms for finding the class and invoking its methods (for example, by invoking **somFindClass, somFindMethod, somLookupMethod, somDispatch, somResolveByName**, and so forth), the resulting client program does not contain any static references to the class library. Thus, SOM will load the class library dynamically during execution of the program. Note: For SOM to be able to load the class library, the **dllname** modifier must be set in the .idl file. (See the topic "Modifier statements" in Chapter 4, "SOM IDL and the SOM Compiler.")

Because the provider of a class library cannot predict which of these ways a class will be used, SOM class libraries must be built such that either usage is possible. The first case above requires the class library to export the entry points needed by the SOM bindings, whereas the second case requires the library to provide an initialization function to create the classes it contains. The following topics discuss each case.

Building export files

The SOM Compiler provides an "exp" emitter for AIX and a "def" emitter for OS/2 or Windows to produce the necessary exported symbols for each class. For example, to generate the necessary exports for a class "A", issue the **sc** or **somc** command with one of the following **–s** options. (For a discussion of the **sc** or **somc** command and options, see "Running the SOM Compiler" in Chapter 4, "SOM IDL and the SOM Compiler.")

For AIX, this command generates an "a.exp" file:

```
sc -sexp a.idl
```

For OS/2, this command generates an "a.def" file:

```
sc -sdef a.idl
```

For Windows, this command generates an "a.def" file:

```
somc -sdef a.idl
```

Typically, a class library contains multiple classes. To produce an appropriate export file for each class that the library will contain, you can create a new export file for the library itself by combining the exports from each "exp" or "def" file into a single file. Following are examples of a combined export “exp” file for AIX and a combined “def” file for OS/2 or Windows. Each example illustrates a class library composed of three classes, “A”, “B”, and “C”.

AIX “exp” file:

```
#! abc.dll
ACClassData
AClassData
ANewClass
BCClassData
BClassData
BNewClass
CCClassData
CClassData
CNewClass
```

OS/2 “def” file:

```
LIBRARY abc INITINSTANCE
DESCRIPTION 'abc example class library'
PROTMODE
DATA MULTIPLE NONSHARED LOADONCALL
EXPORTS
   ACClassData
   AClassData
   ANewClass
   BCClassData
   BClassData
   BNewClass
   CCClassData
   CClassData
   CNewClass
```

Windows “def” file:

```
LIBRARY abc
DESCRIPTION 'abc example class library'
EXETYPE WINDOWS
STUB 'WINSTUB.EXE'
PROTMODE
DATA MOVEABLE SINGLE SHARED LOADONCALL
CODE MOVEABLE DISCARDABLE
HEAPSIZE 4096
STACKSIZE 4096
EXPORTS
   _ACClassData
   _AClassData
   _ANewClass
   _BCClassData
   _BClassData
   _BNewClass
   _CCClassData
   _CClassData
   _CNewClass
```

Other symbols in addition to those generated by the “def” or “exp” emitter can be included if needed, but this is not required by SOM. One feature of SOM is that a class library needs no more than three exports per class (by contrast, many OOP systems require externals for every method as well). One required export is the name of a procedure to create the class (<*className*>**NewClass**), and the others are two external data structures that are referenced by the SOM bindings.

Specifying the initialization function

An initialization function for the class library must be provided to support dynamic loading of the library by the SOM Class Manager. The SOM Class Manager expects that, whenever it loads a class library, the initialization function will create and register class objects for all of the classes contained in the library. These classes are then managed as a group (called an *affinity group*).

One class in the affinity group has a privileged position — namely, the class that was specifically requested when the library was loaded. If that class (that is, the class that caused loading to occur) is subsequently unregistered, the SOM Class Manager will automatically unregister all of the other classes in the affinity group as well, and will unload the class library. Similarly, if the SOM Class Manager is explicitly asked to unload the class library, it will also automatically unregister and free all of the classes in the affinity group.

It is the responsibility of the class-library creator to supply the initialization function. The interface to the initialization function is given by the following C/C++ prototype:

```
#ifdef __IBMC__
  #pragma linkage (SOMInitModule, system)
#endif

SOMEXTERN void  SOMLINK SOMInitModule ( long majorVersion,
                                        long minorVersion,
                                        string className);
```

The parameters provided to this function are the *className* and the major/minor version numbers of the class that was requested when the library was loaded (that is, the class that caused loading). The initialization function is free to use or to disregard this information; nevertheless, if it fails to create a class object with the required name, the SOM Class Manager considers the load to have failed. As a rule of thumb, however, if the initialization function invokes a <*className*>**NewClass** procedure for each class in the class library, this condition will always be met. Consequently, the parameters supplied to the initialization function are not needed in most cases.

Here is a typical class-library initialization function, written in C, for a library with three classes ("A", "B", and "C"):

```
#include "a.h"
#include "b.h"
#include "c.h"
#ifdef __IBMC__
  #pragma linkage (SOMInitModule, system)
#endif

SOMEXTERN void  SOMLINK SOMInitModule (long majorVersion,
                          long minorVersion, string className)
{
    SOM_IgnoreWarning (majorVersion);  /* This function makes  */
    SOM_IgnoreWarning (minorVersion);  /* no use of the passed */
    SOM_IgnoreWarning (className);     /* arguments.   */
    ANewClass (A_MajorVersion, A_MinorVersion);
    BNewClass (B_MajorVersion, B_MinorVersion);
    CNewClass (C_MajorVersion, C_MinorVersion);
}
```

The source code for the initialization function can be added to one of the implementation files for the classes in the library, or you can put it in a separate file and compile it independently.

Using Windows class libraries

Some additional considerations apply for Windows class libraries: Each class library must also supply a Windows LibMain function. The LibMain function is invoked automatically whenever a Windows DLL is loaded, and is responsible for identifying the library and its **SOMInitModule** function to the SOM Kernel. Here is an example of a typical Windows LibMain function for a SOM class library as it would appear in a C or C++ program:

```
#include <som.h>
SOMEXTERN void SOMLINK SOMInitModule (long majorVersion,
                                      long minorVersion,
                                      string className);

#include <windows.h>
int CALLBACK LibMain (HINSTANCE inst,
                      WORD ds,
                      WORD Heapsize,
                      LPSTR cmdLine)
{
    SOM_IgnoreWarning (inst);
    SOM_ignoreWarning (ds);
    SOM_IgnoreWarning (heapSize);
    SOM_IgnoreWarning (cmdLine);

    SOM_ClassLibrary (”xyz.dll”);
    return 1;  /* Indicate success to loader */
}
```

The only operative statement in the LibMain function is the macro **SOM_ClassLibrary**, which identifies the actual file name of the library as it would appear in a Windows LoadLibrary call, and internally generates a reference to the library's **SOMInitModule** function. This information is passed to the SOM Kernel, which in turn registers the library and schedules the execution of the **SOMInitModule** function.

Typically, the SOM Kernel invokes the **SOMInitModule** function of each statically loaded class library during execution of the **SOM_MainProgram** macro within the using application; otherwise, **SOMInitModule** is invoked immediately upon completion of the dynamic loading of a class library by an already executing application. Regardless of the loading mechanism, the SOM Kernel guarantees that the **SOMInitModule** function executes exactly once for each class library.

Creating the import library

Finally, for each of your class libraries, you should create an import library that can be used by client programs (or by other class libraries that use your classes) to resolve the references to your classes.

Here is an example illustrating all of the steps required to create a class library (“abc.dll”) that contains the three classes “A”, “B”, and “C”.

1. Compile all of the implementation files for the classes that will be included in the library. Include the initialization function also.

   For AIX written in **C**:

   ```
   xlc –I. –I$SOMBASE/include –c a.c
   xlc –I. –I$SOMBASE/include –c b.c
   xlc –I. –I$SOMBASE/include –c c.c
   xlc –I. –I$SOMBASE/include –c initfunc.c
   ```

For AIX written in **C++**:

```
xlC -I. -I$SOMBASE/include -c a.C
xlC -I. -I$SOMBASE/include -c b.C
xlC -I. -I$SOMBASE/include -c c.C
xlC -I. -I$SOMBASE/include -c initfunc.C
```

For OS/2 written in **C**:

```
icc -I. -I%SOMBASE%\include -Ge- -c a.c
icc -I. -I%SOMBASE%\include -Ge- -c b.c
icc -I. -I%SOMBASE%\include -Ge- -c c.c
icc -I. -I%SOMBASE%\include -Ge- -c initfunc.c
```

Note: The "–Ge" option is used only with the IBM compiler. It indicates that the object files will go into a DLL.

For OS/2 written in **C++**:

```
icc -I. -I%SOMBASE%\include -Ge- -c a.cpp
icc -I. -I%SOMBASE%\include -Ge- -c b.cpp
icc -I. -I%SOMBASE%\include -Ge- -c c.cpp
icc -I. -I%SOMBASE%\include -Ge- -c initfunc.cpp
```

Note: The "–Ge" option is used only with the IBM compiler. It indicates that the object files will go into a DLL.

For Windows, written in **C**:

```
cl -ALw -G2s -Zp1 -I. -I%SOMBASE%\include -c a.c
cl -ALw -G2s -Zp1 -I. -I%SOMBASE%\include -c b.c
cl -ALw -G2s -Zp1 -I. -I%SOMBASE%\include -c c.c
cl -ALw -G2s -Zp1 -I. -I%SOMBASE%\include -c initfunc.c
```

Notes: If your Windows program uses floats or doubles, also use the "–Oi" flag. These commands are illustrated with the Visual C++ compiler. Sample Makefiles shipped with SOMobjects For Windows also show how to build with other compilers.

For Windows, written in **C++**:

```
cl -ALfu -G2s -Zp1 -I. -I%SOMBASE%\include -c a.cpp
cl -ALfu -G2s -Zp1 -I. -I%SOMBASE%\include -c b.cpp
cl -ALfu -G2s -Zp1 -I. -I%SOMBASE%\include -c c.cpp
cl -ALfu -G2s -Zp1 -I. -I%SOMBASE%\include -c initfunc.cpp
```

Notes: If your Windows program uses floats or doubles, also use the "–Oi" flag. These commands are illustrated with the Visual C++ compiler. Sample Makefiles shipped with SOMobjects For Windows also show how to build with other compilers.

2. Produce an export file for each class.

   For AIX:

   ```
   sc -sexp a.idl b.idl c.idl
   ```

   For OS/2:

   ```
   sc -sdef a.idl b.idl c.idl
   ```

   For Windows:

   ```
   somc -sdef a.idl b.idl c.idl
   ```

3. Manually combine the exported symbols into a single file.

   For AIX, create a file “abc.exp” from “a.exp”, “b.exp”, and “c.exp”. Do *not* include the initialization function (**SOMInitModule**) in the export list.

   For OS/2, create a file “abc.def” from “a.def”, “b.def”, and “c.def”. Include the initialization function (**SOMInitModule**) in the export list so that all classes will be initialized automatically, unless your initialization function does not need arguments and you explicitly invoke it yourself from an OS/2 DLL initialization routine.

   For Windows, create a file “abc.def” from “a.def”, “b.def”, and “c.def”. There is no need to include the **SOMInitModule** function in the export list for Windows programs.

4. Using the object files and the export file, produce a binary class library.

   For AIX:

   ```
   ld –o abc.dll –bE:abc.exp –e SOMInitModule –H512 –T512 \
      a.o b.o c.o initfunc.o –lc –L$SOMBASE/lib –lsomtk
   ```

   The **–o** option assigns a name to the class library (“abc.dll”). The **–bE:** option designates the file with the appropriate export list. The **–e** option designates **SOMInitModule** as the initialization function. The **–H** and **–T** options must be supplied as shown; they specify the necessary alignment information for the text and data portions of your code. The **–l** options name the specific libraries needed by your classes. If your classes make use of classes in other class libraries, include a **–l** option for each of these also. The **ld** command looks for a library named “lib<*x*>.a”, where <*x*> is the name provided with each **–l** option. The **–L** option specifies the directory where the “somtk” library resides.

   For OS/2:

   ```
   set LIB=%SOMBASE%\lib;%LIB%
   link386 /noi /packd /packc /align:16 /exepack \
       a.obj b.obj c.obj initfunc.obj, abc.dll,,os2386 somtk, \
       abc.def
   ```

   If your classes make use of classes in other class libraries, include the names of their import libraries immediately after “somtk” (before the next comma).

   For Windows:

   ```
   link a.obj b.obj c.obj initfunc.obj, abc.dll, nul, \
        ldllcew libw %SOMBASE%\somtk, abc.def
   ```

   Note: /noi should not be used unless all symbols are exported as ordinals.

   If your classes make use of classes in other class libraries, also include the names of their import libraries immediately after “somtk” (before the next comma).

5. Create an import library that corresponds to the class library, so that programs and other class libraries can use (import) your classes.

   For AIX:

   ```
   ar ruv libabc.a abc.exp
   ```
   ◀ Note the use of the “.exp” file, *not* a “.o” file

   The first filename (“libabc.a”) specifies the name to give to the import library. It should be of the form “lib<*x*>.a”, where <*x*> represents your class library. The second filename (“abc.exp”) specifies the exported symbols to include in the import library.

   Caution: Although AIX shared libraries can be placed directly into an archive file (“lib<*x*>.a”), this is not recommended! A SOM class library should have a corresponding import library constructed directly from the combined export file.

For OS/2:

```
implib /noi abc.lib abc.def
```

The first filename ("abc.lib") specifies the name for the import library and should always have a suffix of ".lib". The second filename ("abc.def") specifies the exported symbols to include in the import library. Note: **SOMInitModule** should be included in the <*x*>.dll but not in <*x*>.lib. If you are using an export file that contains the symbol **SOMInitModule**, delete it first; **SOMInitModule** should not appear in your import library because it needs not be exported. **SOMInitModule** should be included when creating your file <*x*>.dll because all classes in the <*x*>.dll will be initialized.

For Windows:

```
implib  abc.lib abc.def
```

The first filename ("abc.lib") specifies the name for the import library and should always have a suffix of ".lib". The second filename ("abc.def") specifies the exported symbols to include in the import library.

5.7 Customizing Memory Management

SOM is designed to be policy free and highly adaptable. Most of the SOM behavior can be customized by subclassing the built-in classes and overriding their methods, or by replacing selected functions in the SOM run-time library with application code. The next five sections contain more advanced topics describing how to customize the various aspects of SOM behavior: memory management, dynamic class loading and unloading, character output, error handling, and method resolution. Information on customizing Distributed SOM is provided in Chapter 6.

The memory management functions used by the SOM run-time environment are a subset of those supplied in the ANSI C standard library. They have the same calling interface and return the equivalent types of results as their ANSI C counterparts, but include some supplemental error checking. Errors detected in these functions result in the invocation of the error-handling function to which **SOMError** points.

The correspondence between the SOM memory-management function variables and their ANSI standard library equivalents is given in Table 1 below.

SOM Function Variable	ANSI Standard C Library Function	Return type	Argument types
SOMCalloc	**calloc( )**	**somToken**	**size_t, size_t**
SOMFree	**free( )**	**void**	**somToken**
SOMMalloc	**malloc( )**	**somToken**	**size_t**
SOMRealloc	**realloc( )**	**somToken**	**somToken, size_t**

Table 1. Memory-Management Functions

An application program can replace SOM's memory management functions with its own memory management functions to change the way SOM allocates memory (for example, to perform all memory allocations as suballocations in a shared memory heap). This replacement is possible because **SOMCalloc, SOMMalloc, SOMRealloc**, and **SOMFree** are actually *global variables* that point to SOM's default memory management functions, rather than being the names of the functions themselves. Thus, an application program can replace SOM's default memory management functions by assigning the entry-point address of the user-defined memory management function to the appropriate global variable. For example, to replace the default free procedure with the user-defined function **MyFree** (which must have the same signature as the ANSI C **free** function), an application program would require the following code:

```
#include <som.h>
/* Define a replacement routine: */
void myFree (somToken memPtr)
{
    (Customized code goes here)
}
...
SOMFree = MyFree;
```

Note: In general, all of these routines should be replaced as a group. For instance, if an application supplies a customized version of **SOMMalloc**, it should also supply corresponding **SOMCalloc**, **SOMFree**, and **SOMRealloc** functions that conform to this same style of memory management.

5.8 Customizing Class Loading and Unloading

SOM uses three routines that manage the loading and unloading of class libraries (referred to here as DLLs). These routines are called by the **SOMClassMgrObject** as it dynamically loads and registers classes. If appropriate, the rules that govern the loading and unloading of DLLs can be modified, by replacing these functions with alternative implementations.

Customizing class initialization

The **SOMClassInitFuncName** function has the following signature:

string (*SOMClassInitFuncName) ( );

This function returns the name of the function that will initialize (create class objects for) all of the classes that are packaged together in a single class library. (This function name applies to all class libraries loaded by the **SOMClassMgrObject**.) The SOM-supplied version of **SOMClassInitFuncName** returns the string `"SOMInitModule"`. The interface to the library initialization function is described under the topic “Creating a SOM Class Library” earlier in this chapter.

Customizing DLL loading

To dynamically load a SOM class, the **SOMClassMgrObject** calls the function pointed to by the global variable **SOMLoadModule** to load the DLL containing the class. The reason for making public the **SOMLoadModule** function (and the following **SOMDeleteModule** function) is to reveal the boundary where SOM touches the operating system. Explicit invocation of these functions is never required. However, they are provided to allow class implementors to insert their own code between the operating system and SOM, if desired. The **SOMLoadModule** function has the following signature:

long (*SOMLoadModule) (string *className***,**
string *fileName***,**
string *functionName***,**
long *majorVersion***,**
long *minorVersion***,**
somToken ***modHandle*);**

This function is responsible for loading the DLL containing the SOM class *className* and returning either the value zero (for success) or a nonzero system-specific error code. The output argument *modHandle* is used to return a token that can subsequently be used by the DLL-unloading routine (described below) to unload the DLL. The default DLL-loading routine returns the DLL’s *module handle* in this argument. The remaining arguments are used as follows:

Argument	**Usage**
fileName	— The file name of the DLL to be loaded, which can be either a simple name or a full path name.
functionName	— The name of the routine to be called after the DLL is successfully loaded by the **SOMClassMgrObject**. This routine is responsible for creating the class objects for the class(es) contained in the DLL. Typically, this argument has the value `"SOMInitModule"`, which is obtained from the function **SOMClassInitFuncName** described above. If no **SOMInitModule** entry exists in the DLL, the default DLL-loading routine looks in the DLL for a procedure with the name *<className>***NewClass** instead. If neither entry point can be found, the default DLL-loading routine fails.

majorVersion — The major version number to be passed to the class initialization function in the DLL (specified by the *functionName* argument).

minorVersion — The minor version number to be passed to the class initialization function in the DLL (specified by the *functionName* argument).

An application program can replace the default DLL-loading routine by assigning the entry point address of the new DLL-loading function (such as *MyLoadModule*) to the global variable **SOMLoadModule**, as follows:

```
#include <som.h>
/* Define a replacement routine: */
long myLoadModule (string className, string fileName,
                   string functionName, long majorVersion,
                   long minorVersion, somToken *modHandle)
{
    (Customized code goes here)
}
...
SOMLoadModule = MyLoadModule;
```

Customizing DLL unloading

To unload a SOM class, the **SOMClassMgrObject** calls the function pointed to by the global variable **SOMDeleteModule**. The **SOMDeleteModule** function has the following signature:

long (*SOMDeleteModule) (in somToken *modHandle***);**

This function is responsible for unloading the DLL designated by the *modHandle* parameter and returning either zero (for success) or a nonzero system-specific error code. The parameter *modHandle* contains the value returned by the DLL loading routine (described above) when the DLL was loaded.

An application program can replace the default DLL-unloading routine by assigning the entry point address of the new DLL-unloading function (such as, *MyDeleteModule*) to the global variable **SOMDeleteModule**, as follows:

```
#include <som.h>
/* Define a replacement routine: */
long myDeleteModule (somToken modHandle)
{
    (Customized code goes here)
}
...
SOMDeleteModule = MyDeleteModule;
```

5.9 Customizing Character Output

The SOM character-output function is invoked by all of the SOM error-handling and debugging macros whenever a character must be generated (see “Debugging” and “Exceptions and error handling” in Chapter 3, “Using SOM Classes in Client Programs”). The default character-output routine, pointed to by the global variable **SOMOutCharRoutine**, simply writes the character to “stdout,” then returns 1 if successful, or 0 otherwise.

For convenience, **SOMOutCharRoutine** is supplemented by the **somSetOutChar** function. The **somSetOutChar** function enables each task to have a customized character output routine, thus it is often preferred for changing the output routine called by **somPrintf** (because **SOMOutCharRoutine** would remain in effect for subsequent tasks). On Windows, the **somSetOutChar** function is required (rather than **SOMOutCharRoutine**); it is optional on other operating systems.

An application programmer might wish to supply a customized replacement routine to:

- Direct the output to **stderr,**
- Record the output in a log file,
- Collect characters and handle them in larger chunks,
- Send the output to a window to display it,
- Place the output in a clipboard, or
- Some combination of these.

With **SOMOutCharRoutine,** an application program would use code similar to the following to install the replacement routine:

```
#include <som.h>
#pragma linkage(myCharacterOutputRoutine, system)
/* Define a replacement routine: */
int SOMLINK myCharacterOutputRoutine (char c)
{
    (Customized code goes here)
}
...
/* After the next stmt all output */
/* will be sent to the new routine   */
SOMOutCharRoutine = myCharacterOutputRoutine;
```

With **somSetOutChar**, an application program would use code similar to the following to install the replacement routine:

```
#include <som.h>
static int irOutChar(char c);

static int irOutChar(char c)
{
    (Customized code goes here)
}

main (...)
{
    ...
    somSetOutChar((somTD_SOMOutCharRoutine *) irOutChar);
}
```

5.10 Customizing Error Handling

When an error occurs within any of the SOM-supplied methods or functions, an error-handling procedure is invoked. The default error-handling procedure supplied by SOM, pointed to by the global variable **SOMError**, has the following signature:

void (*SOMError) (int *errorCode*, **string** *fileName*, **int** *lineNum***);**

The default error-handling procedure inspects the *errorCode* argument and takes appropriate action, depending on the last decimal digit of *errorCode* (see “Exceptions and error handling” in Chapter 3, “Using SOM Classes in Client Programs,” for a discussion of error classifications). In the default error handler, fatal errors terminate the current process. The remaining two arguments (*fileName* and *lineNum*), which indicate the name of the file and the line number within the file where the error occurred, are used to produce an error message.

An application programmer might wish to replace the default error handler with a customized error-handling routine to:

- Record errors in a way appropriate to the particular application,
- Inform the user through the application’s user interface,
- Attempt application-level recovery by restarting at a known point, or
- Shut down the application.

An application program would use code similar to the following to install the replacement routine:

```
#include <som.h>
/* Define a replacement routine: */
void myErrorHandler (int errorCode, string fileName, int lineNum)
{
    (Customized code goes here)
}
...
/* After the next stmt all errors     */
/* will be handled by the new routine */
SOMError = myErrorHandler;
```

When any error condition originates within the classes supplied with SOM, SOM is left in an internally consistent state. If appropriate, an application program can trap errors with a customized error-handling procedure and then resume with other processing. Application programmers should be aware, however, that all methods within the SOM run-time library behave *atomically*. That is, they either succeed or fail; but if they fail, partial effects are undone wherever possible. This is done so that all SOM methods remain usable and can be re-executed following an error.

5.11 Customizing Mutual Exclusion Services (Thread Safety)

The SOM kernel and the other SOMobjects frameworks (DSOM, Persistence, Replication, and so on), have been made *thread safe* with respect to multi-threaded processes. As used here, "thread safe" means that the SOMobjects run time has been implemented using *mutual exclusion semaphores*, or "mutex semaphores," to protect sections of the code which must only be executed by a single thread in a multi-threaded application process at one time.

Some operating systems provide native multi-threading (for example, OS/2). On other operating systems that do not support native multi-threading (such as, AIX 3.2), thread support may be provided as part of particular programming environments (like DCE) or libraries.

It is vital that SOM employ the mutex services that are provided by the thread package used by the application. Consequently, SOM provides a mechanism for defining and customizing mutex services.

Five mutex service functions are used to implement mutual exclusion in SOM. These functions are called indirectly via the global pointer variables defined below. A **somToken** parameter (called "`sem`" below) is used as a generic "handle" to refer to a mutex semaphore — usually it is a pointer to a mutex semaphore variable or data structure. The actual representation of the mutex semaphore is hidden by the functions.

```
unsigned long (*SOMCreateMutexSem)(somToken *sem);
```

The referenced function creates a mutex semaphore, whose handle is returned as an output parameter in the **somToken** variable identified by "`sem`".

If the call succeeds, a 0 is returned. Otherwise, a non-zero error code is returned.

```
unsigned long (*SOMRequestMutexSem)(somToken sem);
```

The referenced function requests ownership of the mutex semaphore identified by the parameter, `sem`. If the semaphore is not currently owned by another thread, ownership is assigned to the calling thread. Otherwise, the calling thread is blocked until the semaphore is released by the current owner.

Important: If the same thread calls **SOMRequestMutexSem** multiple times, a reference count must be kept, so that the semaphore is released only after the same number of calls to **SOMReleaseMutexSem**. Some, but not all, thread packages provide reference counting automatically, via "counting semaphores."

If the call succeeds, a 0 is returned. Otherwise, a non-zero error code is returned.

```
unsigned long (*SOMReleaseMutexSem)(somToken sem);
```

The referenced function releases ownership of the mutex semaphore identified by the parameter, `sem`.

Important: If the same thread calls **SOMRequestMutexSem** multiple times, a reference count must be kept, so that the semaphore is released only after the same number of calls to **SOMReleaseMutexSem**. Some, but not all, thread packages provide reference counting automatically, via "counting semaphores."

If the call succeeds, a 0 is returned. Otherwise, a non-zero error code is returned.

```
unsigned long (*SOMDestroyMutexSem)(somToken sem);
```

The referenced function destroys the a mutex semaphore identified by the parameter, `sem`.

If the call succeeds, a 0 is returned. Otherwise, a non-zero error code is returned.

```
unsigned long (*SOMGetThreadId)();
```

The referenced function returns a small integer identifier for the calling thread. The ID cannot be associated with any other thread in the process. The ID is used as an index for table lookups.

If threads are not supported, the function must return 0.

The actual mutex service function prototypes and global variable declarations are found in file “somapi.h”.

If the underlying operating system supports native multi-threading (currently, only OS/2), SOM provides appropriate default mutex service function implementations. On those operating systems that do not support native multi-threading, the default mutex service functions have null implementations.

An application may want to use threading services different from those provided by the underlying operating system (if any); for example, DCE applications will want to use DCE threads. In that case, the default mutex service functions can be replaced by application-defined functions.

An application program would use code similar to the following to install the replacement routines:

```
#include <som.h>
/* Define a replacement routine: */
unsigned long myCreateMutexSem (somToken *sem)
{
    (Customized code goes here)
}
...
SOMCreateMutexSem = myCreateMutexSem;
```

It is important to install custom mutex service functions *before* any SOM calls are made.

5.12 Customizing Multi-threading Services

Although the SOM kernel and the other SOMobjects frameworks allow applications to be multi-threaded, the kernel and frameworks generally do not require or exploit threads themselves. But there are some exceptions: for example, application servers in DSOM can be configured so that each incoming request is executed on a separate thread.

An application may choose to employ "native" multi-threading services provided by the underlying operating system (for example, OS/2). On other operating systems that do not support native multi-threading (such as, AIX 3.2), thread support may be provided as part of particular programming environments (like DCE) or libraries. SOM provides a mechanism that allows an application to define and customize the multi-threading services used by SOMobjects frameworks.

Four thread service functions are defined for use by SOMobjects frameworks. These functions may be called indirectly via the global pointer variables defined below. A **somToken** parameter (called "`thrd`" below) is used as a generic "handle" to refer to a thread — usually it is a pointer to a thread ID or descriptor. The actual representation of the thread handle is hidden by the functions.

```
typedef void somTD_SOMThreadProc(void * data);
```

```
unsigned long (*SOMStartThread)(somToken *thrd,
                                somTD_SOMThreadProc proc,
                                void *data,
                                unsigned long datasz,
                                unsigned long stacksz);
```

The referenced function starts a thread, and returns a thread handle in the **somToken** variable identified by "`thrd`". The thread executes the procedure whose address is specified by the `proc` parameter; the thread procedure takes a single `void*` argument and returns `void`. The `data` parameter passed to **SOMStartThread** is passed on to the thread procedure; the size of the data parameter, in bytes, is given by `datasz`. A stack of `stacksz` bytes will be allocated for the thread.

Note: On OS/2, the thread procedure must be compiled with _Optlink linkage.

If the call succeeds, a 0 is returned. Otherwise, a non-zero error code is returned.

```
unsigned long (*SOMEndThread)(void);
```

The referenced function terminates the calling thread.

If the call succeeds, a 0 is returned. Otherwise, a non-zero error code is returned.

```
unsigned long (*SOMKillThread)(somToken thrd);
```

The referenced function terminates the thread identified by the input parameter `thrd`.

If the call succeeds, a 0 is returned. Otherwise, a non-zero error code is returned.

```
unsigned long (*SOMYieldThread)(void);
```

The referenced function causes the calling thread to yield control to another thread of equal or higher priority.

If the call succeeds, a 0 is returned. Otherwise, a non-zero error code is returned.

```
unsigned long (*SOMGetThreadHandle)(somToken * thrd);
```

The referenced function returns a handle that can be used to identify the calling thread. The handle is returned in the **somToken** variable pointed to by `thrd`.

If the call succeeds, a 0 is returned. Otherwise, a non-zero error code is returned.

The actual mutex service function prototypes and global variable declarations are found in file "somthrd.h".

If the underlying operating system supports native multi-threading (currently, only OS/2), SOM provides appropriate default multi-threading service function implementations. On those operating systems that do not support native multi-threading, the default multi-threading service functions have null implementations.

An application may want to use threading services different from those provided by the underlying operating system (if any); for example, DCE applications will want to use DCE threads. In that case, the default multi-threading service functions can be replaced by application-defined functions.

An application program would use code similar to the following to install the replacement routines:

```
#include <somthrd.h>
/* Define a replacement routine: */
unsigned long myStartThread (somToken *thrd,
                             somTD_SOMThreadProc proc,
                             void *data,
                             unsigned long datasz,
                             unsigned long stacksz)
{
    (Customized code goes here)
}
...
SOMStartThread = myStartThread;
```

It is important to install custom multi-threading service functions *before* any SOM calls are made.